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.

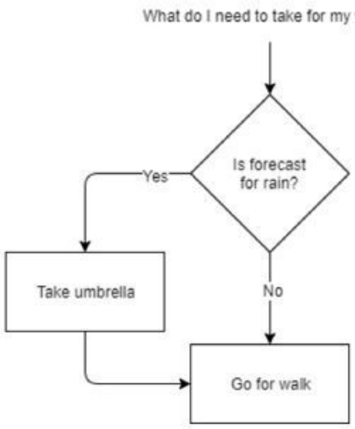
The intent of this thinking skill is to take a divide and conquer approach by taking the one overwhelming whole and break it down into manageable pieces. Additionally the sequencing (ordering) of steps is usually an important consideration.

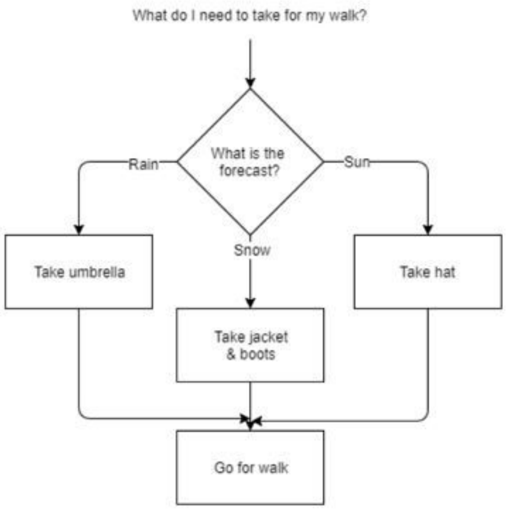
A common everyday example of thinking procedurally is a recipe for a meal.

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?

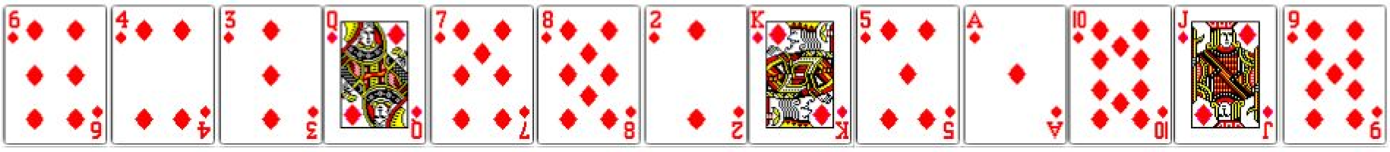
To illustrate what these concepts are, it is best to walk through an example. To get started, these are the steps you are going to follow:

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

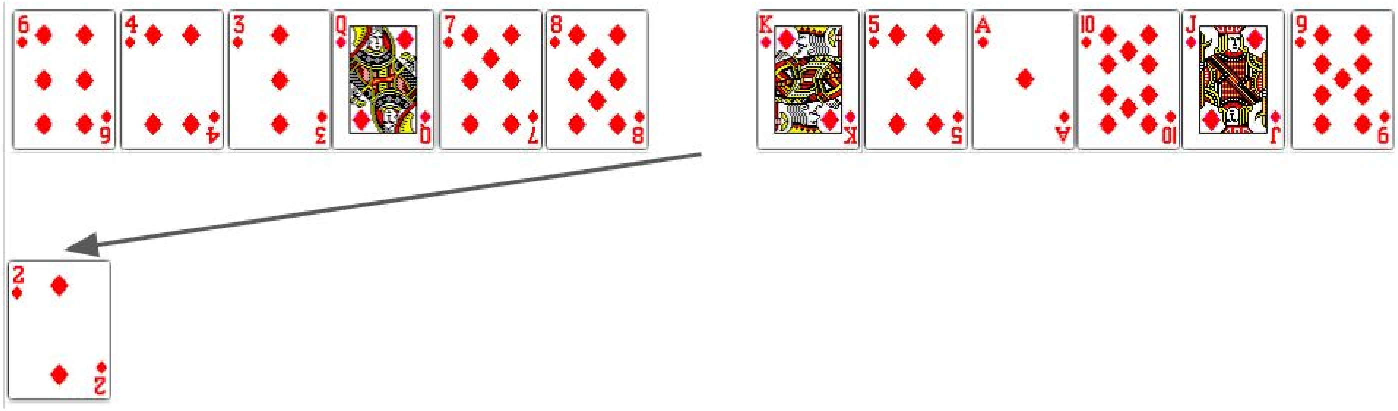
So, by applying step 1, let's create a small, human solvable version of the problem. This is important because an enormous infinitely large set of numbers is too overwhelming to think about. We do know how to sort playing cards though, so let's start with that.



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.

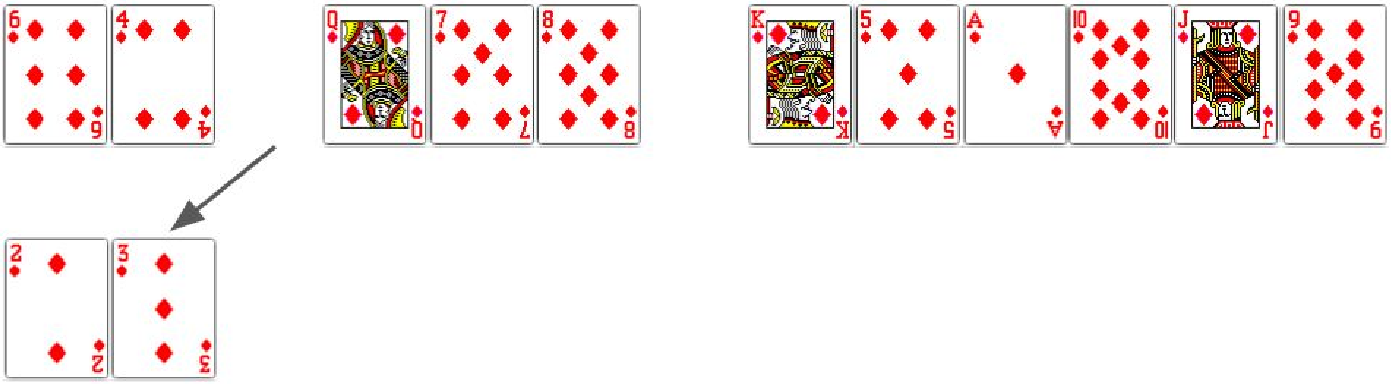
This is my attempt at documenting the human process of sorting cards... see how it compares to what you came up with...

First action...

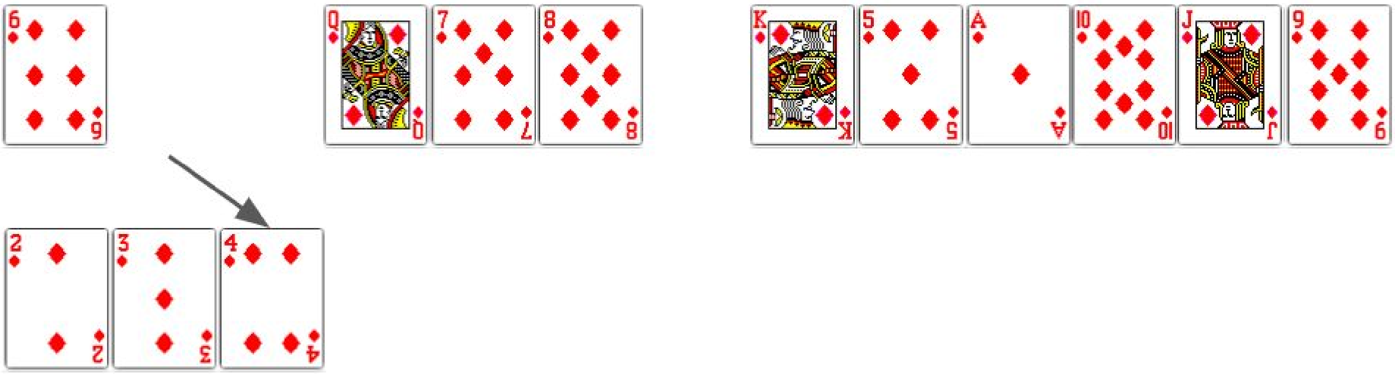


Second action..

.



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.

```python
# 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...

```python
# 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.

```python
# 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.

There is a flaw in the example I shared with you.

Java arrays are not resizable.

- We can't just delete a value from one array
- We can't just add a value to another array
- It breaks the entire logic of the solution.

This time… let's constrain ourselves within the limits of Java. We have an array of 13 values. What system would you use to sort those cards now?

6	4	3	12	7	8	2	13	5	14	10	11	9

So, again, you can loop through them all quickly to find that 2 is the smallest, but how are you going to bring it to the front? You have to keep yourself to the 13 boxes.

Some thoughts:

- Swap the 2 and the 6?
- Hold on to the 2, shuffle everything else down one place until the empty space that used to have the 2 is filled, and then insert the 2 at the front?
- Some other system?

Have a go coming up with a procedure in plain english that others can follow.

Test the procedure on someone who was not involved in writing it.

Once your written procedure works, then have a go at programming it.

2. Program design

Program design is all about solving problems. If you can't solve and articulate the problem by hand, you will not be able to solve it with code! There are three key strategies this course would like you to be familiar with for articulating and testing algorithms. They are:

- Pseudo code
- Flow charts
- Trace tables

2.1 Pseudo code

There are a lot of different computer programming languages available, each serving different needs. Algorithms, however, are universal. A programmer who uses one language, should be able to communicate how to create an algorithm to another programmer who uses a completely different language without knowing anything about it!

For that reason, most of the time an algorithm is being written it won't be in a language-specific format, but one of two generic forms: flow charts or pseudo-code.

Pseudo code is "Structured English".

It's intent: To clearly communicate an algorithm to other programmers regardless of the programming language(s) they are familiar with.

It uses general programming constructs rather than anything language specific.

Since it is not an actual language, there is not a fixed syntax for its use in the broader computer science industry, provided it is generic enough to achieve its aims.

That said, to achieve consistency in iGCSE & IB exam settings, these courses have their own set syntax for the manner in which questions are provided.

An example:

```
EVENS ← 0
input N
loop while N <> 0
    if N modulus 2 == 0 then
        EVENS ← EVENS + 1
    else
        ODDS ← ODDS + 1
    end if
    input N
end loop
output ODDS, EVENS
```

What is the above algorithm doing?

Bonus points: What is the error in this algorithm?

The course requires you to be able to create as well as interpret/analyse pseudo code.

Where answers are to be written in pseudocode, the examiners will be looking for clear algorithmic thinking to be demonstrated. In examinations, this type of question will be written in the approved notation, so a familiarity with it is expected. It is accepted that under exam conditions candidates may, in their solutions, use pseudocode similar to a programming language with which they are familiar. This is acceptable. The mark scheme will be written using the approved notation. Provided the examiners can see the logic in the candidate's response, regardless of language, it will be credited. No marks will be withheld for syntax errors ... for answers to be written in pseudo code

Given the statements made above about there not being a "correct" way to write it, there is a set syntax that the IB will use when they write pseudo code questions for you in the exam. As per the expectations comments above, you are not obliged to follow their syntax in your responses, and you will not lose credit for doing so, but you should be familiar enough with their syntax to be able to properly interpret the questions they give you.

These methods, in their pseudocode format, may be used without explanation or clarification in examination questions. Teachers should ensure that candidates will be able to interpret these methods when presented as part of an examination question.

The following comes from the IB documents

- "Approved notation for developing pseudocode" available at https://pbaumgarten.com/ib-compsci/ib-compsci-pseudocode-flowcharts.pdf
- "Pseudocode in Examinations" available at https://pbaumgarten.com/ib-compsci/ib-compsci-pseudocode-in-detail.pdf

Approved notation for developing pseudocode

When developing pseudocode teachers must use the symbols below, which are those used in mathematics.

This information should be distributed to candidates as close as possible to the commencement of teaching of the course. This notation sheet will be available to candidates during the external examinations.

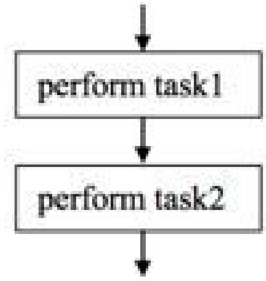
Conventions	Variable names are all capitals, for example, CITY Pseudocode keywords are lower case, for example, loop, if … Method names are mixed case, for example, getRecord Methods are invoked using the "dot notation" used in Java, C++, C#, and similar languages, for example, BIGARRAY.binarySearch(27)

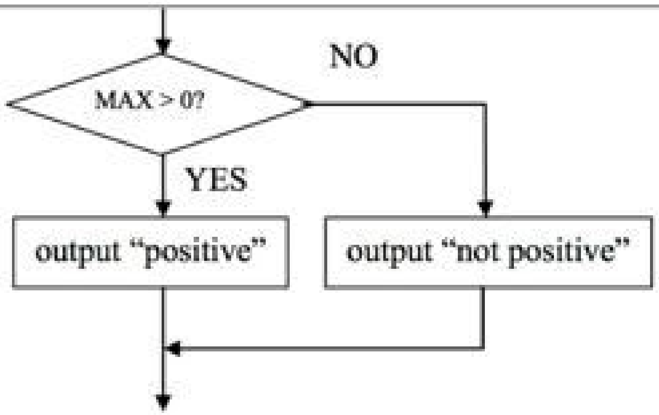
Variable names	These will be provided and comments // used, for example:
	N = 5 // the number of items in the array
	SCOREHISTORY,getExam(NUM) // get the student's score on exam NUM

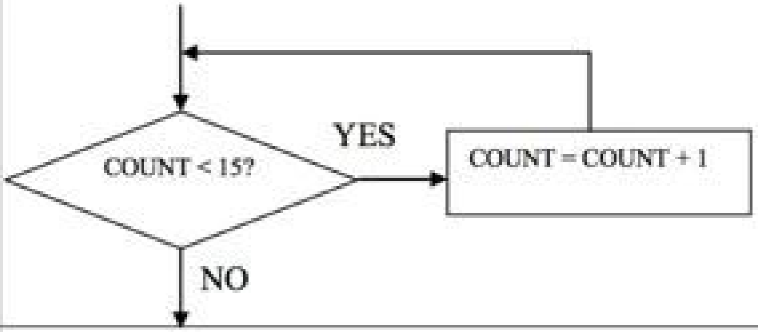
Assigning a value to a variable	Values will be assigned using = , for example:
	N = 5 // indicates the array has 5 data items
	VALUE[0] = 7 // assigns the first data item in the array a value of 7

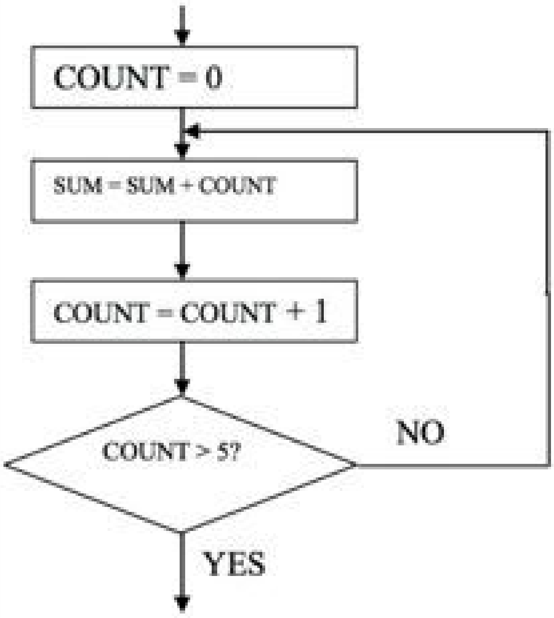
Output of information	Output—this term is sufficient to indicate the data is output to a printer, screen, for example: output COUNT // display the count on the screen

Symbol	Definition	Examples	
=	is equal to	X = 4, X = K	If X = 4
>	is greater than	X > 4	if X > 4 then
>=	is greater than or equal to	X >= 6	loop while X >= 6
<	is less than	VALUE[Y] < 7	loop until VALUE[Y] < 7
<=	is less than or equal to	VALUE[] <=12	if VALUE[Y] <= 12 then
≠	not equal to	X ≠ 4, X ≠ K	
AND	logical AND	A AND B	if X < 7 AND Y > 2 then
OR	logical OR	A OR B	if X < 7 OR Y > 2 then
NOT	logical NOT	NOT A	if NOT X = 7 then
mod	modulo	15 mod 7 = 1	if VALUE[Y] mod 7 = 0 then
div	integer part of quotient	15 div 7 = 2	if VALUE[Y] div 7 = 2 then

Operation	Flowchart example	Pseudocode example
sequential operations		perform task1 perform task2

conditional operations	 MAX > 0?
	NO
	YES
	output "positive"
	output "not positive"
		if MAX > 0 then
		output "positive"
		else
		output "not positive"
		end if

while-loop		loop while COUNT < 15
		COUNT = COUNT + 1
		end loop

from/to-loop	 COUNT = 0
	SUM = SUM + COUNT
	COUNT = COUNT + 1
	COUNT > 5?
	NO
	YES
		loop COUNT from 0 to 5
		SUM = SUM + COUNT
		end loop

Higher Level and Standard Level

Arrays

An array is an indexed and ordered set of elements. Unless specifically defined in the question, the index of the first element in an array is 0.

```
NAMES[0]  // The first element in the array NAMES
```

Strings

A string can contain a set of characters, or can be empty. Strings can be used like any other variable.

```
MYWORD = "This is a string"
if MYWORD = "the" then
    output MYWORD
end if
```

Strings should be regarded as a class of objects. However no methods belonging to that class are part of this standard. Instead, if a specialized method such as `charAt()` or `substring()` is to be used in an examination, it will be fully specified as part of the question in which it is needed.

Collections

Collections store a set of elements. The elements may be of any type (numbers, objects, arrays, Strings, etc.).

A collection provides a mechanism to iterate through all of the elements that it contains. The following code is guaranteed to retrieve each item in the collection exactly once.

```
// STUFF is a collection that already exists
STUFF.resetNext()
loop while STUFF.hasNext()
    ITEM = STUFF.getNext()
    // process ITEM in whatever way is needed
end loop
```

Collections

Method name	Brief description	Example: HOT, a collection of temperatures	Comment

`addItem()`	Add item	`HOT.addItem(42)`
		`HOT.addItem("chile")`
			Adds an element that contains the argument, whether it is a value, String, object, etc.

`getNext()`	Get the next item	`TEMP = HOT.getNext()`	`getNext()` will return the first item in the collection when it is first called. Note: `getNext()` does not remove the item from the collection.

`resetNext()`	Go back to the start of the collection	`HOT.resetNext()`
		`HOT.getNext()`
			Restarts the iteration through the collection. The two lines shown will retrieve the first item in the collection.

`hasNext()`	Test: has next item	`if HOT.hasNext() then`	Returns TRUE if there are one or more elements in the collection that have not been accessed by the present iteration: The next use of `getNext()` will return a valid element.
`isEmpty()`	Test: collection is empty	`if HOT.isEmpty() then`	Returns TRUE if the collection does not contain any elements.

Examples of Pseudocode

AVERAGING AN ARRAY

The array STOCK contains a list of 1000 whole numbers (integers). The following pseudocode presents an algorithm that will count how many of these numbers are non-zero, adds up all those numbers and then prints the average of all the non-zero numbers (divides by COUNT rather than dividing by 1000).

```
COUNT = 0
TOTAL = 0

loop N from 0 to 999
  if STOCK[N] > 0 then
    COUNT = COUNT + 1
    TOTAL = TOTAL + STOCK[N]
  end if
end loop

if NOT COUNT = 0 then
  AVERAGE = TOTAL / COUNT
  output "Average = " , AVERAGE
else
  output "There are no non-zero values"
end if
```

COPYING FROM A COLLECTION INTO AN ARRAY

The following pseudocode presents an algorithm that reads all the names from a collection, NAMES, and copies them into an array, LIST, but eliminates any duplicates. That means each name is checked against the names that are already in the array. The collection and the array are passed as parameters to the method.

```
COUNT = 0   // number of names currently in LIST

loop while NAMES.hasNext()

  DATA = NAMES.getNext()

  FOUND = false
  loop POS from 0 to COUNT-1
    if DATA = LIST[POS] then
      FOUND = true
    end if
  end loop

  if FOUND = false then
    LIST[COUNT] = DATA
    COUNT = COUNT + 1
  end if
end loop
```

FACTORS

The following pseudocode presents an algorithm that will print all the factors of an integer. It prints two factors at a time, stopping at the square root. It also counts and displays the total number of factors.

```
// recall that
//   30 div 7 = 4
//   30 mod 7 = 2

NUM = 140  // code will print all factors of this number
F = 1
FACTORS = 0

loop until F*F > NUM  //code will loop until F*F is greater than NUM
  if NUM mod F = 0 then

    D = NUM div F
    output NUM , " = " , F , "*" , D

    if F = 1 then
      FACTORS = FACTORS + 0
    else if F = D then
      FACTORS = FACTORS + 1
    else
      FACTORS = FACTORS + 2
    end if

  end if
  F = F + 1
end loop
output NUM , " has " , FACTORS , " factors "
```

COPYING A COLLECTION INTO AN ARRAY IN REVERSE

The following pseudocode presents an algorithm that will read all the names from a collection, `SURVEY`, and then copy these names into an array, `MYARRAY`, in reverse order.

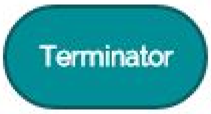
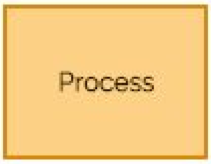
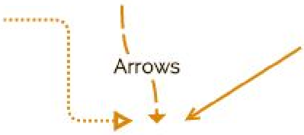
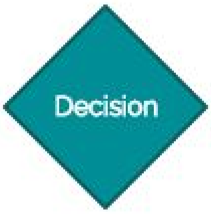
```
// MYSTACK is a stack, initially empty

COUNT = 0 // number of names

loop while SURVEY.hasNext()
  MYSTACK.push( SURVEY.getNext() )
  COUNT = COUNT + 1
end loop

// Fill the array, MYARRAY, with the names in the stack

loop POS from 0 to COUNT-1
  MYARRAY[POS] = MYSTACK.pop()
end loop
```

2.2 Flow charts

IB exams will not require you to create your own flow charts but they will present flow charts to you for analysis and interpretation.

Flow charts are usually quite intuitive. The main issue is to correctly interpret the symbols. Some guidelines are:

- Always format your flow from left to right or top to bottom.
- Run your return lines under your flowchart, making sure that they don't overlap.
- Maintain consistent spacing between symbols.
- Use the correct symbol for each step (diamond shapes are for decisions, rectangles are used for processes, start/end shapes should be the same, etc.)

Symbol	Meaning
	Start or end point of a program.
	A task or action that needs to be performed.
	The "flow" indicating what part of the program should be executed next.
	A decision is required to move forward. This could be a binary, this-or-that choice or a more complex decision with multiple choices. Make sure that you capture each possible choice within your diagram.

From:
https://cacoo.com/blog/keep-it-simple-how-to-avoid-overcomplicating-your-flowcharts/
https://www.gliffy.com/blog/how-to-flowchart-basic-symbols-part-1-of-3

2.3 Trace tables

A trace table is a method of performing a manual dry run on an algorithm where you perform the computations. It is useful for error checking of simple algorithms.

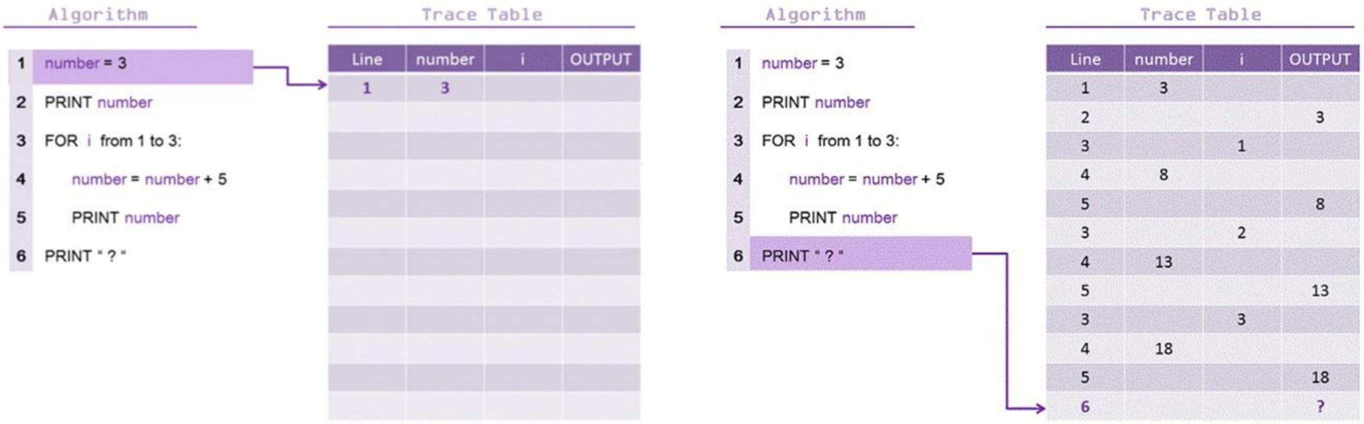
To make one, draw a table of columns, one for each variable. Then walk through the algorithm by hand, writing any changes to the data line by line using test data.

To properly test an algorithm with a trace table it is important to use a good variety of test data, both normal and erroneous data.

You are expected to be able to analyse and create trace tables for different algorithms. Some questions you could be asked to perform with trace tables include:

- Create a trace table to identify the error within a given algorithm.
- Determine the number of times a step in a given algorithm will be performed for given input data
- Suggest changes in an algorithm that would improve efficiency, for example, using a flag to stop a search immediately when an item is found
- Justify an algorithms efficiency, correctness, reliability or flexibility

An example



Algorithm

```
1  number = 3
2  PRINT number
3  FOR i from 1 to 3:
4      number = number + 5
5      PRINT number
6  PRINT " ? "
```

Trace Table

Line	number	i	OUTPUT
1	3		

Algorithm

```
1  number = 3
2  PRINT number
3  FOR i from 1 to 3:
4      number = number + 5
5      PRINT number
6  PRINT " ? "
```

Trace Table

Line	number	i	OUTPUT
1	3		
2			3
3		1	
4	8		
5			8
3		2	
4	13		
5			13
3		3	
4	18		
5			18
6			?

2.4 Practice

1. Write pseudo code that will sum all the even numbers input from a user. Stop after 10 numbers have been input.

2. Write pseudo code that reads in any three numbers and outputs them into sorted order.

3. Design an algorithm... where the user continually inputs a number, stopping when -1 is provided. For each number, the count and the sum of the numbers provided is kept and output at the end.

4. Design an algorithm... that counts numbers. Have the user input two positive integers, and the program counts up by increments of 1 from the smaller number up to but not including the larger number.

5. Write pseudo code that will perform the following:

   Read in 5 separate numbers.
   Calculate the average of the five numbers.
   Find the smallest (minimum) and largest (maximum) of the five entered numbers.
   Output the results found from steps 2 and 3.

6. Write pseudo code that will calculate a running sum. A user will enter numbers that will be added to the sum and when a negative number is encountered, stop adding numbers and write out the final result.

7. The Hailstone problem. The Hailstone Series is generated using the following high level algorithm:

   Pick a positive number (0 or greater)
   If it is odd, triple the number and add one.
   If it is even, divide the number by two.
   Go back to step 2.
   This series will eventually reach the repeating "ground" state: 4, 2, 1, 4, 2, 1

   Here is the sequence generated for an initial value of 26:

   26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1

   Your task is to convert the high-level algorithm into pseudo-code. At the end, display how many items are in the sequence and what is the largest number computed in the sequence. You can assume the "ground" state commences when the integer 4 is computed. At this stage you can terminate computation of the series.

   Once complete, produce a trace table for your Hailstone Series algorithm, given an input of 17.

8. Fizz buzz. This is a "famous" programming job interview question. Create the pseudo code for a Fizz Buzz generator, and use a trace table for values up to 15.

   Write a program that prints the numbers from 1 to 100.
   For multiples of three print "Fizz" instead of the number

For the multiples of five print "Buzz".
For numbers which are multiples of both three and five print "FizzBuzz"
Example output: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, ...

9. An IB question. The following exercise comes from the May 2015 IB exam. Trace the following algorithmic fragment for N = 6. Show all working in a trace table.

```
SUM = 0
loop COUNT from 1 to (N div 2)
    if N mod COUNT = 0 then
        SUM = SUM + COUNT
    end if
end loop
if SUM = N then
    output "perfect"
else
    output "not perfect"
end if
```

Want more practice?

Pseudocode past paper questions for practice
https://pbaumgarten.com/ib-compsci/unit-4/assets/pseudocode-past-questions.pdf

70 pseudocode questions for practice
https://pbaumgarten.com/igcse-compsci/distribute/pseudocode-70-questions.pdf
(I have the answers for these if you are interested)

3. Standard algorithms

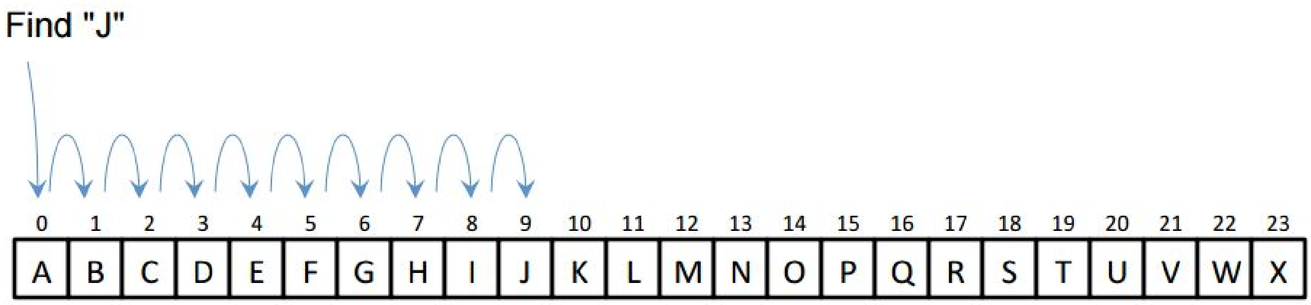
There are a number of algorithms the IB course will assume you know by memory. These are the "standard algorithms".

- Sequential search
- Binary search
- Bubble sort
- Selection sort

Any algorithm not defined as a "standard algorithm" can be considered a "novel algorithm" and will be either presented to you in the exam or is an algorithm you would be expected to devise.

3.1 Sequential search

Sometimes also known as linear search, the premise of the sequential search is to iterate through a set of data sequentially until you find the item you are looking for.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X

The pseudocode is as simple as a for loop containing an if statement.

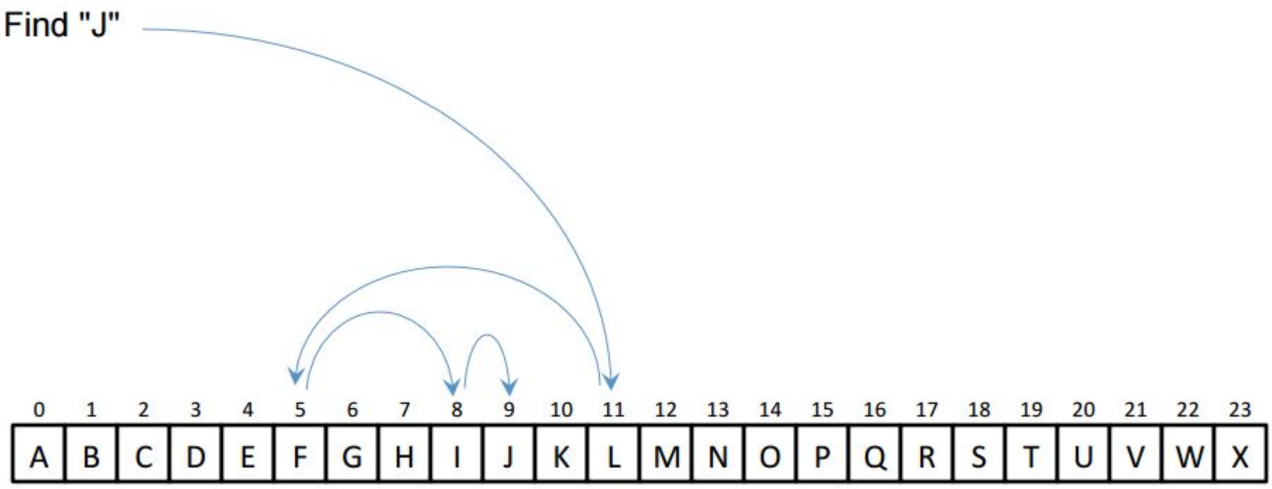
```
// javascript
function sequential( haystack, needle ) {
    for (var i=0; i<haystack.length; i++) {
        if (needle == haystack[i]) {
            return(i);
        }
    }
    return(-1);
}
```

One interesting thing to note with the sequential search is it does not require your data to be sorted ahead of the search. This could save a lot of processing time. If, however, your data is already sorted, you can include within your sequential search a test to see if we have already gone past the point at which the record we are looking for would exist. In that case, we can abort the remainder of the search and return a "not found" result.

("find j" images from https://qccs.me/classes/cs313/fried/website/searching.html)

3.2 Binary search

A binary search divides a range of values into halves, and continues to narrow down the field of search until the unknown value is found. It is the classic example of a "divide and conquer" algorithm. Your data must be pre-sorted!



Check this excellent illustration of binary vs sequential search!

- https://pbaumgarten.com/ib-compsci/unit-4/binary-and-linear-search-animations.gif

The pseudocode for a binary search (iterative version)

```
// javascript
function binary( haystack, needle ) {
    var max = haystack.length-1;
    var min = 0;
    while (max >= min) {
        var mid = Math.floor((min+max)/2);
        if (haystack[mid] == needle) {
            return mid;
        }
        if (haystack[mid] > needle) {
            max = mid - 1;
        }
        if (haystack[mid] < needle) {
            min = mid + 1;
        }
    }
    return (-1);
}
```

This is known as an iterative binary search. There is also a recursive binary search which we will look at in a later unit. Just be aware if you have to google binary search that there are two types.

3.3 Search practice

Question 1: Tracing binary search

Create a trace table on the binary search algorithm above with the following values:

haystack = [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
1st needle = 15
2nd needle = 5
Compare that on a trace table for a sequential search algorithm.

It should be fairly simple to code an implementation of this to test your algorithms - have a go.

Question 2: Simple spell checker

If a user inputs a sentence, separate the individual words, strip out any punctuation, check each word against your dictionary list. Print each word out with PASS or FAIL against it based on if it was found in your dictionary.

Devise an algorithm for this and have a go at implementing it. Will you use the sequential or binary search? Why?

You will need a "word list" to use as your dictionary. There are lots of good word lists available online. This is one I've used before but feel free to find your own.

- https://github.com/dolph/dictionary/blob/master/popular.txt

3.4 Bubble sort

Check this animation of how bubble sort works:

- http://pbaumgarten.com/ib-compsci/unit-4/bubblesort.gif

Here is a walkthrough of bubble sort in action

```
[ 19 32 11 21 39 14 42 25 ] Original

[ 19 32 11 21 39 14 42 25 ] Compare
[ 19 32 11 21 39 14 42 25 ] Compare
[ 19 11 32 21 39 14 42 25 ] Swap
[ 19 11 32 21 39 14 42 25 ] Compare
[ 19 11 21 32 39 14 42 25 ] Swap
[ 19 11 21 32 39 14 42 25 ] Compare
[ 19 11 21 32 39 14 42 25 ] Compare
[ 19 11 21 32 14 39 42 25 ] Swap
[ 19 11 21 32 14 39 42 25 ] Compare
[ 19 11 21 32 14 39 42 25 ] Compare
[ 19 11 21 32 14 39 25 42 ] Swap

[ 19 11 21 32 14 39 25 42 ] Compare
[ 11 19 21 32 14 39 25 42 ] Swap
[ 11 19 21 32 14 39 25 42 ] Compare
[ 11 19 21 32 14 39 25 42 ] Compare
[ 11 19 21 32 14 39 25 42 ] Compare
[ 11 19 21 14 32 39 25 42 ] Swap
[ 11 19 21 14 32 39 25 42 ] Compare
[ 11 19 21 14 32 39 25 42 ] Compare
[ 11 19 21 14 32 25 39 42 ] Swap
[ 11 19 21 14 32 25 39 42 ] Compare

[ 11 19 21 14 32 25 39 42 ] Compare
[ 11 19 21 14 32 25 39 42 ] Compare
[ 11 19 21 14 32 25 39 42 ] Compare
[ 11 19 14 21 32 25 39 42 ] Swap
[ 11 19 14 21 32 25 39 42 ] Compare
[ 11 19 14 21 32 25 39 42 ] Compare
[ 11 19 14 21 25 32 39 42 ] Swap
[ 11 19 14 21 25 32 39 42 ] Compare
[ 11 19 14 21 25 32 39 42 ] Compare

[ 11 19 14 21 25 32 39 42 ] Compare
[ 11 19 14 21 25 32 39 42 ] Compare
[ 11 14 19 21 25 32 39 42 ] Swap
[ 11 14 19 21 25 32 39 42 ] Compare
[ 11 14 19 21 25 32 39 42 ] Compare
[ 11 14 19 21 25 32 39 42 ] Compare
[ 11 14 19 21 25 32 39 42 ] Compare
[ 11 14 19 21 25 32 39 42 ] Compare

[ 11 14 19 21 25 32 39 42 ] Sorted
```

The bubble sort algorithm pseudocode looks like

```
while (swap_occurred):
   swap_occurred = false
   for every value in the array [1]:
      compare value to its neighbour
      if neighbour > value:
         swap
         set swap_occurred = true
      end-if
   End-for
end-while
return array

---
[1] except the last one
```

3.5 Selection sort

Check this animation of how selection sort works:

- http://pbaumgarten.com/ib-compsci/unit-4/selectionsort.gif

A walk through of Selection sort in action...

```
[ 19 32 11 21 39 14 42 25 ] Original

[ 19 32 11 21 39 14 42 25 ] Start
[ 19 32 11 21 39 14 42 25 ] Compare
[ 19 32 11 21 39 14 42 25 ] Potential swap
[ 19 32 11 21 39 14 42 25 ] Compare
[ 19 32 11 21 39 14 42 25 ] Compare
[ 19 32 11 21 39 14 42 25 ] Compare
[ 19 32 11 21 39 14 42 25 ] Compare
[ 19 32 11 21 39 14 42 25 ] Compare
[ 11 32 19 21 39 14 42 25 ] Swap

[ 11 32 19 21 39 14 42 25 ] Move to next
[ 11 32 19 21 39 14 42 25 ] Potential swap
[ 11 32 19 21 39 14 42 25 ] Compare
[ 11 32 19 21 39 14 42 25 ] Compare
[ 11 32 19 21 39 14 42 25 ] Potential swap
[ 11 32 19 21 39 14 42 25 ] Compare
[ 11 32 19 21 39 14 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Swap

[ 11 14 19 21 39 32 42 25 ] Move to next
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Compare

[ 11 14 19 21 39 32 42 25 ] Move to next
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Compare

[ 11 14 19 21 39 32 42 25 ] Move to next
[ 11 14 19 21 39 32 42 25 ] Potential swap
[ 11 14 19 21 39 32 42 25 ] Compare
[ 11 14 19 21 39 32 42 25 ] Potential swap
[ 11 14 19 21 25 32 42 39 ] Swap

[ 11 14 19 21 25 32 42 39 ] Move to next
[ 11 14 19 21 25 32 42 39 ] Compare
[ 11 14 19 21 25 32 42 39 ] Compare

[ 11 14 19 21 25 32 42 39 ] Move to next
[ 11 14 19 21 25 32 42 39 ] Potential swap
[ 11 14 19 21 25 32 39 42 ] Swap

[ 11 14 19 21 25 32 39 42 ] Sorted
```

The pseudo code for the selection sort looks like...

```
for every value in the array [1]:
   for every value to the right:
      compare current_value to right_value
      if a right_value < current_value:
         save index location of right_value
      end-if
   end-for
   if an index location was saved:
      swap current_value and right_value
   end-if
end-for
return array

---
[1] except the last one
```

Comparing bubble sort & selection sort

Both of these sorting algorithms look very inefficient (and, generally speaking they are) but they do have unique edge-case advantages when they might be of real-world use.

Bubble sort is highly efficient when the dataset is already mostly sorted. It is ideal to use when you want to add a new value into it's sorted position to an already sorted dataset.

Selection sort is optimised for when writing data is very expensive (slow) when compared to reading. Eg:. writing to flash memory or EEPROM. No other sorting algorithm has less data movement! This is important to realise as you can be asked questions justifying the use of one algorithm over another.

3.6 Sort practice

Question 1

Here is a set of 50 integers from https://www.random.org/integer-sets/...

```
34, 43, 73, 88, 9, 91, 48, 10, 94, 3, 75, 87, 74, 63, 11, 36, 82, 100, 28, 68, 18, 60, 35, 81, 79, 23, 86, 41, 49, 2, 7, 83, 6, 58, 47, 39, 27, 54, 21, 12, 4, 5, 31, 46, 62, 55, 37, 57, 67, 93
```

Implement both sorting algorithms to process your data set. Compare and contrast how many comparison read operations each takes (ie: count the number of times the if statement operates), and how many write operations each takes (ie: how many times the array is written to).

Is it correct that the selection sort orders of magnitude more efficiently for the number of write operations?

```
100, 1, 2, 3, 5, 9, 10, 11, 12, 17, 20, 23, 25, 31, 33, 35, 39, 40, 42, 43, 44, 45, 46, 47, 51, 52, 55, 56, 59, 61, 62, 63, 64, 66, 69, 70, 75, 77, 78, 79, 80, 81, 83, 86, 87, 88, 89, 92, 94, 96, 98
```

Swapping to use the second set of numbers, is it true that the bubble sort is more efficient for this "almost sorted" set? By how much?

Question 2

Once you are successfully sorting numbers, how about sorting strings such as a list of names?

```
"Eustolia","Nathan","Milissa","Willie","Hoyt","Alexandria","Clelia","Alpha","Delbert","Boyd","Milton","Vivan","Constance","Hilma","Irving","Carie","Nicky","Adele","Carlene","Hermina","Ayana","Frederica","Arianna","Zandra","Vina","Lory","Mao","Alona","Lajuana","Coralie","Allyson","Corey","Geraldo","Sherryl","Monika","Charlesetta","Deon","Coletta","Jed","Carlee","Lise","Teresita","Odelia","Adeline","Olive","Elisha","Casey","Octavia","Alexandra","Franklyn"
```

From http://listofrandomnames.com/.

Question 3

Sort the given set of dates given in dd/mm/yyyy format into their correct calendar order so the date which occurs first, appears first in the list.

```
"29/06/2009","06/06/1984","16/06/1993","23/11/1996","23/09/1986","07/07/2002","29/01/1999","13/06/1998","14/02/2005","29/08/2013","24/12/2009","04/09/2019","02/02/2020","22/10/2015","08/11/1987","23/10/2018","14/10/2015","19/02/2013","05/06/1989","21/08/1991","06/06/2005","03/02/1993","01/12/1993","01/09/1995","24/01/2018"
```

4. Algorithm efficiency

We have compared the efficiency of our sorting algorithms. Studying algorithm efficiency and having a language to describe it is an important part of the science of Computer Science.

From the IB CS syllabus:

Students should understand and explain the difference in efficiency between a single loop, nested loops, a loop that ends when a condition is met or questions of similar complexity. Students should also be able to suggest changes in an algorithm that would improve efficiency, for example, using a flag to stop a search immediately when an item is found, rather than continuing the search through the entire list. Examination questions will involve specific algorithms (in pseudocode/flowcharts), and students may be expected to give an actual number (or range of numbers) of iterations that a step will execute.

As you can see from the above, it is important to understand the efficiency of different algorithms. One measure used in industry is called Big O notation.

Big-O describes the relationship of how the runtime will scale with respect to certain input variables.

Some common examples of Big-O expressions:

`O(1)`	constant
`O(log(n))`	logarithmic
`O(n)`	linear
`O(n^2)`	quadratic
`O(n^c)`	polynomial
`O(c^n)`	exponential

If the run time will increase linearly with respect to the size of the input data, this would be `O(n)`. An example is to loop through the values of an array.

If the run time will increase exponentially with respect to the size of the input data, this would be `O(n^2)`. An example is to have a loop instead a loop, where both iterate through the values of an array.

A good introduction to Big O notation can be found with the following video:

HackerRank (2016), Big O notation
https://www.youtube.com/watch?v=v4cd1O4zkGw

There were four important rules from the video:

Rule 1 - If you have two important steps in your algorithm, you add those steps.

```
function something() {
    doTask1()       // O(a)
    doTask2()       // O(b)
}
// Overall result = O(a+b)
```

Rule 2 - Drop constants.

```
function something() {
    for each item in array:     // O(n)
        min = MIN(item, min)
    for each item in array:     // O(n)
        max = MAX(item, max)
}
// The overall run time will still increase linearly, so it is O(n) not O(2n).
```

Rule 3 - Different inputs usually use different variables to represent them in the O() relationship.

```
function something() {
    for each item in A:
       total = total + item
    for each item in B:
       total = total + item
    return total
}
// The overall runtime would be O(a*b)
```

Rule 4 - Drop non-dominant terms.

For example, if an algorithm as a nested for-loop which would be O(n^2) and a regular for-loop O(n), it is not O(n + n^2) because the O(n) is going to be completely dominated by the O(n^2) to such a degree that it is meaningless to worry about.

Here is also a great analogy from the comments section of that video: Let's say you're making dinner for your family. O is the process of following a recipe, and n is the number of times you follow a recipe.

- `O(1)` - you make one dish that everyone eats whether they like it or not. You follow one recipe from top to bottom, then serve (1 recipe). <-- How I grew up
- `O(n)` - you make individual dishes for each person. You follow a recipe from top to bottom for each person in your family (recipe times the number of people in your family).
- `O(n^2)` - you make individual dishes redundantly for every person. You follow all recipes for each person in your family (recipe times the number of people squared).
- `O(log n)` - you break people into groups according to what they want and make larger portions. You make one dish for each group (recipe times request)

For a graphical comparison that illustrates the difference in processing load of different O() algorithms, check the https://bigocheatsheet.com website.

4.1 Big-O Practice

What is the Big O in the following?

Question 1

```
boolean containsValue(int[] list, int val) {
   for (int item : list) {
      if (item==val) {
         return true;
      }
   }
   return false;
}
```

Question 2

```
boolean containsDuplicates(int[] a, int[] b) {
   for (int x : a) {
      for (int y : b) {
         if (x==y) { return true; }
      }
   }
   return false;
}
```

Question 3

```
int fibonacci(int n) {
   if (n <= 1) { return n; }
   int fib = 1;
   int prev = 1;
   for (int i=2; i<n; i++) {
      int temp = fib;
      fib += prev;
      prev = temp}
   return fib
}
```

Question 4

```
boolean isFirstElementZero(int[] a) {
   if (a[0] == 0) {
      return true;
   } else {
      return false;
   }
}
```

Question 5

Which is algorithmically more efficient?

You are maintaining a customer contact details database for a sales business, currently with 1+ million records. On any given day, several hundred customers will notify a change of address for the database. Additionally, on any given day, the support staff will receive several thousand calls through the switchboard, where they will have to look up the customer data.

Should your program...?

- Save new changes as they occur, and have the support desk app sequentially search through the records? ... or ...
- Bubble sort after every save, and have the support desk app binary search through the records.

Justify your choice with reference to the O(n) of each method. [5 marks]

Question 6

A supermarket inventory system needs to be updated to keep an accurate record of the various stock on hand. The supermarket carries a range of about 10 000 different items on its shelves. The hard disk of the computer used is efficient at performing data lookups, but takes about 100 times longer to write to disk than a lookup.

With approximately 1000 transactions per day, would it be more efficient to:

- Selection sort the data after each individual sale? or
- Bubble sort the data after closing each day?

5. Programming concepts

5.1 Conventions and best practice

Meaningful identifiers, proper indentation and adequate comments all improve the readability of code and thus save money, time and effort in programming teams. Pragmatically as a student of this course, it is in your interests for your code to be quickly and easily understood by your teacher and exam markers.

Code layout

Coding should follow tabbed indentations in blocks between braces.

Comments

Commenting should be clear, consistent, and regular.

Naming consistency

Names should be meaningful! When naming your identifiers, Java conventions are as follows

- Class name: Start with uppercase and be a noun. Animal, Genus etc.
- Method name: Start with uppercase letter eg. addToList, initialiseStack etc.
- Variable name: Start with lowercase letter eg. firstName, orderNumber etc.
- Package name: Lowercase letters data structures, GUIs etc.
- Constants name: Uppercase letter. RED, MAX_PRIORITY

Variables

- Since variables describe an attribute or property, they are like generally adjectives or nouns.
- A noun based naming scheme generally works best.

Functions

- A function should perform one job.
- Minimise side effects. This means the function should not alter data outside of it.
- Since a function performs an action, name it using a verb.
- Names should be descriptive and consistent.
- Outputs should be avoided, use return whenever possible

5.2 Pass by value / pass by reference

Functions or subroutines are a useful way of allowing a program to be modularised / divided into sub tasks. It is frequently the case that you will want your function to receive particular information that it acts on, and then have it return a result. The passing of information is generally referred to as passing arguments or parameters.

Programming languages have two different methods of passing parameters, and the method used can significantly affect how your program behaves. These two methods are pass by value and pass by reference.

```
function doubleIt( var number ) {
    number = number * 2;
    return( number );
}

function main() {
    var a = 10;
    b = doubleIt( a );
    print( a )
    print( b )
}
```

What will the above program print?

The answer is... it depends on if the programming language is using pass by reference or pass by value!

Pass by value will take the value stored within a, allocate new memory space for the number, and place a copy of the number 10 into that new memory space. Pass by reference will lookup the memory address used by a, and assign the variable number the same memory address.

Now... What will the above program print in each case?

Technically Java uses pass by value for everything. In practice, it will only seem like it behaves that way for primitive data types (integers, floats, booleans etc). For objects and complex data structures such as arrays, it will seem like it is using pass by reference. This is because internally within Java, the "value" of an array or object variable name is the memory address of where the array or object is stored.

The implication of this is that this will behave like pass by reference even though it is technically pass by value.

```
import java.util.Arrays;

public static int[] doubleIt(int[] data) {
    for (int i=0; i<data.length; i++) {
        data[i] = data[i] * 2;
    }
    return(data);
}
```

```
public static void main(String[] args) {
    int[] numbers = { 1, 2, 3, 4, 5 };
    int[] numbers2 = doubleIt( numbers );
    System.out.println( Arrays.toString( numbers ));
    System.out.println( Arrays.toString( numbers2 ));
}
```

One solution to get around this problem in the above example would be to change the `doubleIt()` function as follows...

```
public static int[] doubleIt(int[] data) {
    int[] result = new int[data.length];
    for (int i=0; i<data.length; i++) {
        result[i] = data[i] * 2;
    }
    return(result);
}
```

Why does this solve the problem? *discuss*

It is important to be on guard for this issue and be aware of how it can impact your programming. When starting to get serious with any new programming language, knowing what is treated as pass by value, and what is treated as pass by reference is a question you will need to know the answer to.

5.3 Understanding code

How do computers understand code?
https://www.youtube.com/watch?v=QXjU9qTsYCc

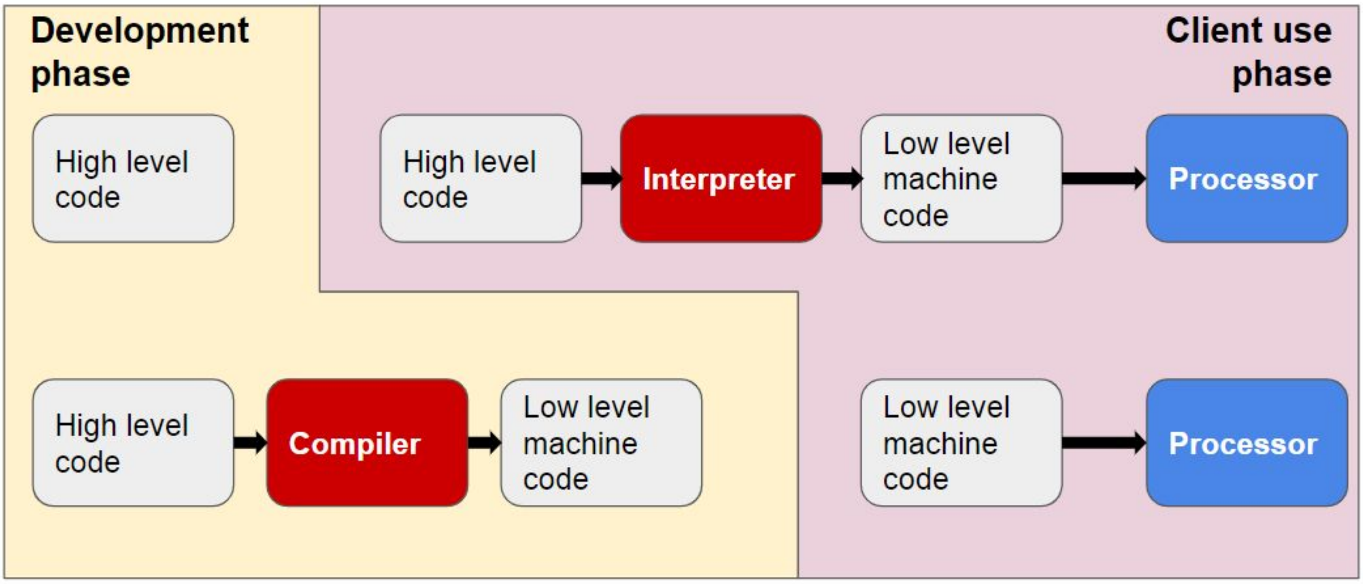
Software programs are sets of instructions. For a CPU to execute these instructions, each one must first be translated into machine code – simple binary codes that activate parts of the CPU.

The CPU only performs a few basic functions:

- performing mathematical operations like addition and subtraction
- moving data from one memory location to another
- making decisions and jumps to a new set of instructions based on those decisions

A piece of software, such as a game or web browser, combines these functions to perform more complex tasks. These are known as compound operations.

There are two main ways that our programs are converted into this machine code: interpretation and compilation.



Compiler

A compiler translates a human-readable program directly into an executable, machine-readable form before the program can run.

Interpreter

An interpreter translates a human-readable program into an executable, machine-readable form, instruction by instruction. It then executes each translated instruction before moving on to the next one. A program is translated every time it is run. Python is an example of an interpreted language.

5.4 High / low level languages

As we know, a computer program is a list of instructions that enable a computer to perform a specific task. The languages that we use to write our computer programs in are generally split into two categories, depending on the task and the hardware being used. These are high level languages and low level languages.

High Level Languages

When we think about computer programmers, we are probably thinking about people who write in high-level languages.

High level languages are written in a form that is close to our human language, enabling the programmer to just focus on the problem being solved.

No particular knowledge of the hardware is needed as high level languages create programs that are portable and not tied to a particular computer or microchip.

These programmer friendly languages are called 'high level' as they are far removed from the machine code instructions understood by the computer.

Examples include: C++, Java, Pascal, Python, Visual Basic.

Advantages

- Easier to modify as it uses English like statements
- Easier/faster to write code as it uses English like statements
- Easier to debug during development due to English like statements
- Portable code – not designed to run on just one type of machine

Modern high level languages would come with features such as:

- Variables & constants
- Types - integers, floating point numbers, characters, strings, booleans
- Operators - add, subtract, multiply, divide, modulus, concatenate etc
- Loops - for, while, repeat until
- Branching - if, if else, else
- Collections - saving and retrieving multiple items to one variable identifier
- Arrays/lists - a subtype of collection
- Sub routines, functions
- Error or exception handling

Low Level Languages

Low level languages are used to write programs that relate to the specific architecture and hardware of a particular type of computer. They are *closer* to the native language of a computer (binary), making them harder for programmers to understand. Low level languages include Assembly Language and Machine Code.

Assembly Language

- Few programmers write programs in low level assembly language, but it is still used for developing code for specialist hardware, such as device drivers.
- It is easily distinguishable from a high level language as it contains few recognisable human words but plenty of mnemonic code.
- Typical assembly language opcodes include: add, subtract, load, compare, branch, store

Advantages

- Can make use of special hardware or special machine-dependent instructions (e.g. on the specific chip)
- Translated program requires less memory
- Write code that can be executed faster
- Total control over the code
- Can work directly on memory locations

Machine Code

- Programmers rarely write in machine code (binary) as it is difficult to understand.