

Unit 4.1: Computational thinking

Jeanette Wing (Vice President of Microsoft Research, and previously President's Processor of Computer Science at Carnegie Mellon University, Pittsburgh) wrote a short but highly influential paper outlining the importance of computational thinking. That paper forms the basis of this unit within the IB course.

You should read it... <https://www.cs.cmu.edu/~15110-s13/Wing06-ct.pdf>

There are competing thoughts as to how best categorize computational thinking processes. The IB uses these six:

- Thinking procedurally
- Thinking logically
- Thinking ahead
- Thinking concurrently
- Thinking abstractly
- Thinking recursively (HL)

Others, such as Google and Code.org use the following alternative labels:

- Decompose – Break a big problem into it's constituent parts.
- Patterns – Identify what pieces have in common, and what remains distinct
- Abstraction – Create an abstraction around the patterns
- Algorithms – Write an algorithm so that resolves are easy to achieve.

They all achieve the same point – a set of thinking skills that guides you through solving a problem.

Thinking procedurally

- Turning the solution to your problem into a set of steps that can be followed.
- Following those steps should reproduce the correct solution each time.
- Each step may call upon a sub-procedure with its own list of steps.
- Divide and conquer – take the one overwhelming whole and break it down into manageable pieces.
- Sequencing (ordering) of steps is usually an important consideration.

Example: Cleaning the entire house/apartment, against cleaning the kitchen, bathroom, living room, bedrooms

- Tackle it one task at a time, and eventually the whole will be done.
- Each task identified can also be broken down into sub-tasks. Using the cleaning analogy, this could be vacuum, dust, sweep, put rubbish in the bin, pick clothes up from the floor.
- One important consideration is the order of tasks. Look back at that list of cleaning tasks - is that the order that will achieve the cleanest bedroom? What should the order be?

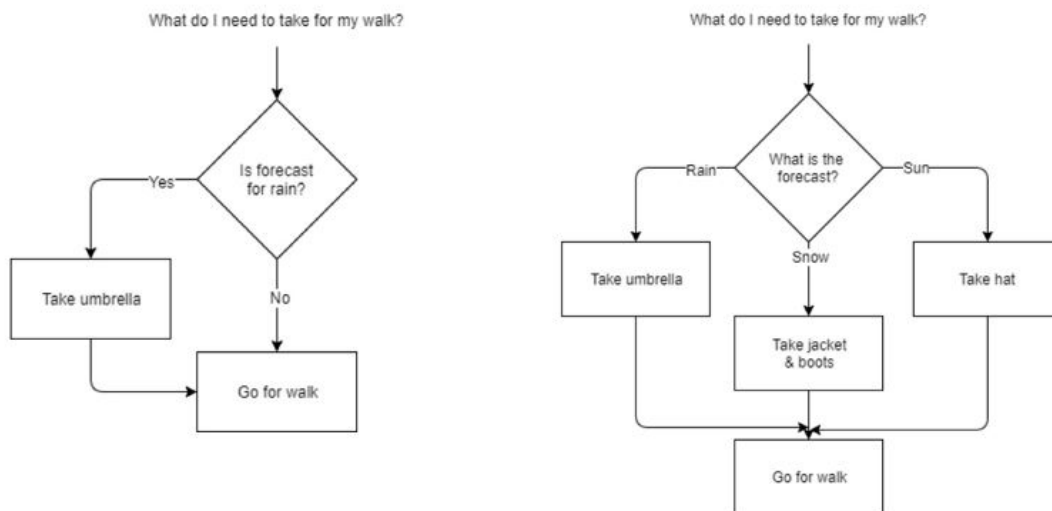
Finally, Look for familiar things. Do not reinvent the wheel. If a solution exists, use it. If you have solved the same or a similar problem before, just repeat the successful solution. We usually do not consciously think, "I have seen this before, and I know what to do" - we just do it. Humans are good at recognizing similar situations. We do not have to learn how to go to the store and buy milk, then to buy eggs, then to buy candy. We know that going to the store is always the same and only what we buy is different.

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.

Decisions may be multi faceted. Compare these two:



The question we ask ourselves in the "if" statement is known as the condition. We can use "boolean logic" to create multiple conditions for a single "if". Both of these do the same thing:

```
if (time is 6:00am) and (day is weekday) then
  wake up for school
end if
```

```
if (time is 6:00am) and ((day is Monday) or (day is Tuesday) or
  (day is Wednesday) or (day is Thursday) or (day is Friday)) then
  wake up for school
end if
```

Logical decisions can also be applied to things of a repetitive nature

```
while (hungry) then
  raid pantry
  eat
end
```

```
for (each episode of Game Of Thrones)
  watch the episode
end
```

Activity: Logical thinking with a flow chart

On 1st June 2009, Air France flight 447 left Rio de Janeiro heading to Paris. It was a routine international flight. In the early hours of the morning, over the Atlantic Ocean, contact was lost, and the aeroplane vanished.

On investigation, the plane showed signs of a high-speed impact with water as the nose cone was flattened. This ruled out a bomb or structural break-up. It was determined that the plane crashed into the water due to pilot error.

The plane flew through a thunderstorm. Other aeroplanes had diverted that night, as is standard practice in bad weather. The pitot tubes (speed sensors) had frozen over as a result. This caused the autopilot to switch off and incorrect readings to be sent to the cockpit. This is expected behaviour, and pilots are trained to recognise this. Believing that the plane was losing altitude, the pilot pulled back on the stick to raise the nose, in an attempt to gain height. The instruments continued to show the plane falling. If an aircraft's nose is pointed up too far, it loses speed, causing the engines to stall. The correct action is to point the nose down, gaining speed, before levelling off.

With the aid of a flowchart, show how logical thinking could have avoided this accident.

Activity: Film canister sort

You will be divided into groups. Each group will receive 8 film canisters and a balance scale. Your task is to sort the canisters from lightest to heaviest.

Use your procedural and logical thinking skills to devise the most efficient instructions for sorting the canisters. Count the number of "comparisons" your procedure makes. The winning team are the ones with the least comparisons (most efficient algorithm)

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:

- Pre-ordering.
- Shopping lists. You know you want to bake a cake, so you make sure you have all the necessary ingredients ahead of time.

- Pre-heating 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.

Activity: Die Hard movie challenge

Can you solve the puzzle in Die Hard?

- Introducing the problem (play up to 1:05)
- Solution (play from 1:05)

<https://www.youtube.com/watch?v=6cAbgAaEOVE>

Activity: Egg drop

You are given two eggs, and access to a 100-storey building. Both eggs are identical. The aim is to find out the highest floor from which an egg will not break when dropped out of a window from that floor. If an egg is dropped and does not break, it is undamaged and can be dropped again. However, once an egg is broken, that's it for that egg.

If an egg breaks when dropped from floor n , then it would also have broken from any floor above that. If an egg survives a fall, then it will survive any fall shorter than that.

The question is: What strategy should you adopt to minimize the number egg drops it takes to find the solution?. (And what is the worst case for the number of drops it will take?)

Spoilers ahead! Don't read on further than necessary. If you do read a hint, stop and give it a genuine attempt before reading on to the next one

**** Hint 1 ****

Whilst it's not strictly part of the puzzle, let's first imagine what we'd do if we had only one egg.

Once this egg is broken, that's it, no more egg. So, we really have no other choice but to start at floor 1. If it survives, great, we go up to floor 2 and try again, then floor 3 ... all the way up the building; one floor at a time. Eventually the egg will break* and we'll have a solution. For example, if it breaks on floor 57, we know that the highest floor that an egg can withstand a drop from is floor 56.

There's no other one egg solution. If we'd been feeling lucky we could have gone up the floors in two's but imagine if the egg broke on floor 16; we have no way of knowing if it would have also broken on floor 15!

**** Hint 2 ****

At the other extreme, what if we had an infinite number of eggs? (Or at least as many eggs as we need). What would our strategy be here? In this case we'd use one of a programmer's favorite tools, the binary tree.

First we'd go to floor 50 and drop an egg. It either breaks, or it does not. The outcome of this drop instantly cuts our problem in half. If it breaks, we know the solution lives in the bottom half of the building (floor 1 – floor 49). If it survives, we know the solution is in the top half of the building (floor 51 – 100). On each drop, we keep dividing the problem in half and half again until we get to our solution.

The mathematicians in the audience will quickly see that the number of drops required for this solution is $\log_2 n$, where n is the number of floors of the building. (This is like asking how many powers of two there are). ie: $\log_2 100 = 6.644$, or 7.

**** Hint 3 ****

It does not take much imagination to see why a binary search solution will not work (optimally) for two eggs. Let's imagine we did try a binary search and dropped our first egg from floor 50. If it broke, we'd be instantly reduced to a one egg problem.

What happens if we started off with our first egg going up by floors ten at a time? We can start dropping our egg from floor 10; if it survives we try floor 20, then floor 30 ... we carry on until the first egg breaks. Once we've broken our first egg we know that the solution must lay somewhere in the nine floors just below, so we back off nine floors and step through these floors one at a time until we find a solution.

The question really comes down to: what is the optimal number of floors to skip with the first egg?

**** Hint 4 ****

What we need is a solution that minimizes our maximum regret. We need is a strategy that tries to make solutions to all possible answers the same depth (same number of drops). The way to reduce the worst case is to attempt to make all cases take the same number of drops.

**** Solution ****

Did you work it out? The solution is at <https://code.oursky.com/famous-egg-dropping-puzzle-in-combinatorics/>

Activity: Ball bearings

You have 10 boxes of ball bearings (each ball weighing exactly 10 gm) with one box with defective ball bearings (each one of the defects weigh 9 gm). You are given an electronic weighing machine and only one chance at it. How will find out which box has the defective ball bearings?

Thinking concurrently

Concurrency = dealing with multiple things happening at the same time.

Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously, instead of sequentially as they would have to be carried out by a single processor. (Oracle)

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)

Activity: Multi theaded sorting

How long (comparisons) does it take to sort a deck of cards with:

- 1 thread (1 person)

- 2 threads (2 people)
- 4 threads (4 people)?

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.

Video: [Robotics Academy \(2016\) Abstraction - Computational Thinking](#) (2:28)

Real world examples:

- Maps – An abstraction contain 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, etc

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.

Exercises

Jake & Jill's weekly food shop

Jake and Jill are quite fed up of how long they spend in the supermarket each week doing their weekly food shop. They decide what they want when they are actually walking around the supermarket and they often have to go back multiple times in the week as they run out of items. This method of shopping is also resulting in a very expensive total weekly shopping bill!

How could they use the principles of computational thinking to make their weekly shopping experience as efficient as possible. Their overall aims are to:

- Spend as little time as possible in the supermarket each week
- Save as much money as possible

Taxi driver

A taxi driver uses his experience, a GPS navigation system and radio tuned to traffic information to work out how to get passengers from A to B.

In what ways is the taxi driver able to:

- Think abstractly
- Think ahead
- Think logically

Unit 4.2: 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!

Documenting program design

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

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.

What is pseudo code?

It 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 it's use in broader computer science industry, provided it is generic enough to achieve it's aims.

That said, to achieve consistency in iGCSE & IB, these courses do have a fixed syntax for the manner in which questions are provided.

Pseudo code example

```

ODDS ← 0
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?

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. You could, for instance, be asked to:

- 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

Algorithm		Trace Table			
		Line	number	i	OUTPUT
1	number = 3	1	3		
2	PRINT number				
3	FOR i from 1 to 3:				
4	number = number + 5				
5	PRINT number				
6	PRINT " ? "				

Algorithm		Trace Table			
		Line	number	i	OUTPUT
1	number = 3	1	3		
2	PRINT number	2			3
3	FOR i from 1 to 3:	3		1	
4	number = number + 5	4	8		
5	PRINT number	5			8
6	PRINT " ? "	3		2	
		4	13		
		5			13
		3		3	
		4	18		
		5			18
		6			?

What would a trace table look like for the previous pseudo code algorithm?

```

ODDS ← 0
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

```

Odds	Evens	N	Output
.			
.			
.			
.			
.			
.			
.			
.			
.			
.			

Questions: Pseudo code & trace tables

Q1. 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.

Q2. Design an algorithm... that counts numbers. Have the user inputs two positive integers, and the program counts up by increments of 1 from the smaller number up to but not including the larger number.

Q3. Design an algorithm... where the user inputs three peoples names, and the program prints the name of the person whose name is longest (contains the most characters).

Q4. Design an algorithm... where the user inputs three numbers, and then outputs them in order from lowest to highest value.

IB Pseudo code and flow charts

You need 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 markscheme 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.

IB exams will not require you to create your own flow charts but they will present flow charts to you for analysis and interpretation. The IB format for these is included in the document that follows.

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>

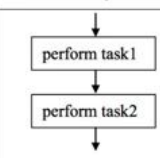
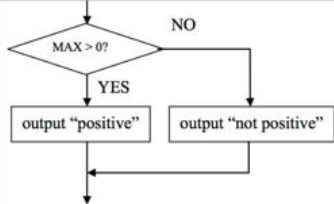
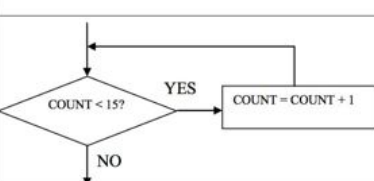
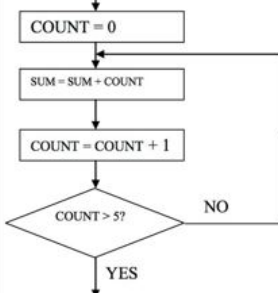
Approved notation for developing pseudocode

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		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		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
<code>addItem()</code>	Add item	<code>HOT.addItem(42)</code> <code>HOT.addItem("chile")</code>	Adds an element that contains the argument, whether it is a value, String, object, etc.
<code>getNext()</code>	Get the next item	<code>TEMP = HOT.getNext()</code>	<code>getNext()</code> will return the first item in the collection when it is first called. Note: <code>getNext()</code> does not remove the item from the collection.
<code>resetNext()</code>	Go back to the start of the collection	<code>HOT.resetNext()</code> <code>HOT.getNext()</code>	Restarts the iteration through the collection. The two lines shown will retrieve the first item in the collection.
<code>hasNext()</code>	Test: has next item	if <code>HOT.hasNext()</code> 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 <code>getNext()</code> will return a valid element.
<code>isEmpty()</code>	Test: collection is empty	if <code>HOT.isEmpty()</code> 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
```

Exercices for pseudo code

Question 1

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

Question 2

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

Question 3

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.

Question 4

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.

Question 5: 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

Task 1:

- Convert the high-level algorithm into pseudo-code
- At the end, display how many items are in the sequence
- At the end, display 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.

Task 2: Produce a trace table for your Hailstone Series algorithm, given an input of 17.

Question 6: Fizz buzz

This is a "famous" programming job interview question.

- 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, ...

Task: Create the pseudo code for a Fizz Buzz generator, and use a trace table to for values up to 15.

Question 7: 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
```

More questions

The following past paper questions include those that require use arrays and collections. Make sure you are confident with those for your exams.

- [Pseudocode past paper questions for practice](#)
- [70 pseudocode practice questions](#) (I have the solutions for these for you to self-check)

The 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.

- [Bubble sort, insertion sort and quick sort demo video](#) (4m38 TED-Ed)

Sequential search

Search through a set of data sequentially until you find the item you are looking for.

```
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.

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!

```
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);  
}
```

Note: 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.

Search algorithm exercises

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 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.

Use the 7 step process to devise an algorithm for this and have a go at implementing it if you wish. 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>

Bubble sort

Bubble sort: The highest value "bubbles" to the top each round.

```
function bubbleSort( data ) {  
  var done = false;  
  while (!done) {  
    done = true;  
    for (var i=1; i<data.length; i++) {  
      if (data[i] < data[i-1]) {  
        done = false;  
        var temp = data[i-1];  
        data[i-1] = data[i];  
        data[i] = temp;  
      }  
    }  
  }  
  return(data);  
}
```


Selection sort

The pseudo code for a selection sort

```
function selectionSort( data ) {
  for (var i=0; i<data.length; i++) {
    var indexOfLowest = i;
    for (var j=i; j<data.length; j++) {
      if (data[j] < data[indexOfLowest]) {
        indexOfLowest = j;
      }
    }
    var temp = data[i];
    data[i] = data[indexOfLowest];
    data[indexOfLowest] = temp;
  }
  return(data);
}
```

Comparing Bubble and Selection sort

To see an animation of either sort in action look at <https://visualgo.net/bn/sorting>

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 its 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.

Sort algorithm exercises

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 efficient 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","Ay  
ana","Frederica","Arianna","Zandra","Vina","Lory","Mao","Alona","Lajuana","Coralie","Allyson"  
,"Corey","Geraldo","Sherryl","Monika","Charlesetta","Deon","Coletta","Jed","Carlee","Lise","T  
eresita","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","1  
3/06/1998","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"
```

From <https://www.random.org/calendar-dates/>

Algorithm efficiency and Big-O

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 run time 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 inside 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](#)

There were four important rules from the video:

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)
```

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).
```

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)
```

4. Drop non-dominant terms.

For example, if an algorithm has 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 bigocheatsheet.com website.

Questions: Determine the Big O

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?

Unit 4.3: Programming concepts

The following is not an introduction to a programming language per-se, rather a discussion about the conceptual ideas involved in programming.

Nature of programming languages

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.

Do how do computers read code?

This video is a look at the link between programming languages, compilers, assembler and machine code - <https://www.youtube.com/watch?v=QXjU9qTsYCc> (12:00)

Low level v higher level languages

A computer program is a list of instructions that enable a computer to perform a specific task.

Computer programs can be written in high and low level languages, depending on the task and the hardware being used.

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

High Level Languages

High level languages are written in a form that is close to our human language, enabling to 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

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 easy 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.

Interpretation v 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.

Features of modern programming languages

Regardless of the programming language you learn, there are core constructs that are generally available in all major, modern programming languages. Be sure you are aware of the terminology involved.

- 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

Unit review

Computer Science Illuminated by Nell Dale & John Lewis (page numbers based on 6th edition):

- Problem solving strategies: End of chapter 4 (page 240), exercises 1-15
- Programming concepts: End of chapter 4 (page 240), exercises 16-33 (exlcude recursion questions)
- Algorithm design: End of chapter 4 (page 240), exercises 40-62