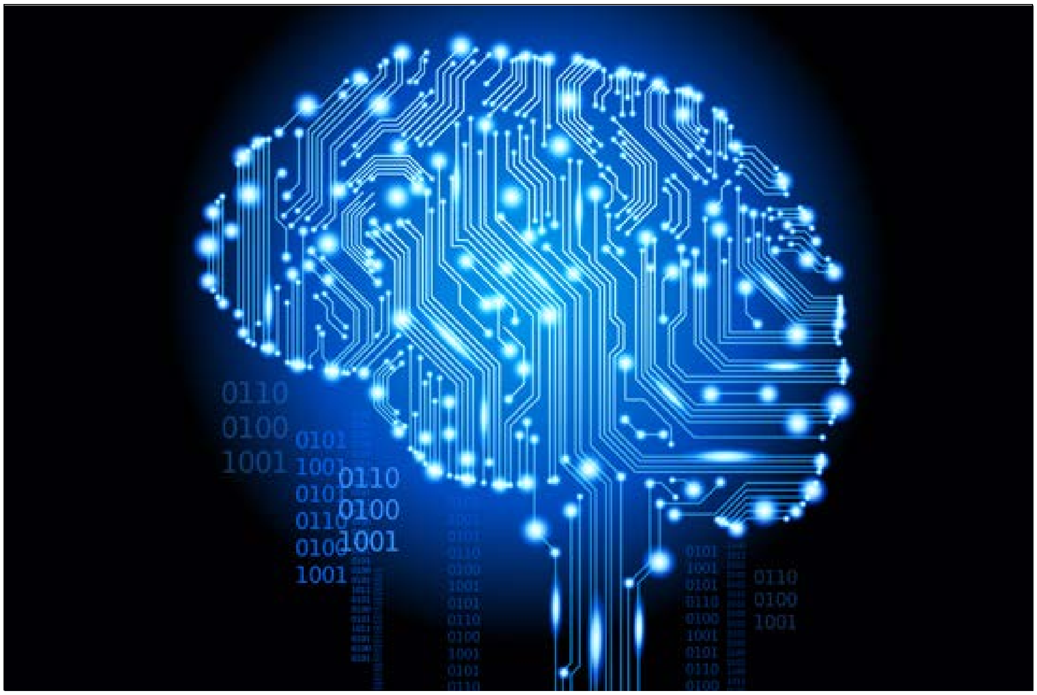
Topic 4: Computational Thinking, Problem Solving and Programming



Programming

Pseudocode

Flowcharts

4.1 General principles

4.1.1- 4.1.3 Thinking procedurally

Identify the procedure appropriate to solving a problem.

Evaluate whether the order in which activities are undertaken will result in the required outcome. Explain the role of sub-procedures in solving a problem.

Thinking procedurally involves having the decomposed parts of a problem available. Following that you are going to look at how these can be ordered procedurally in order to create a program or algorithm.

The outcome of thinking procedurally is to:

- Identify the components of a problem.
- Identify the components of a solution to a problem.
- Determine the order of steps needed to solve a problem.
- Identify sub-procedures necessary to solve a problem.

Example - Customer in a restaurant orders a cup of tea.

Step 1 - Identify the components of a problem
I need to make a cup of tea.

Step 2 - Identify the components of the solution to the problem
Pour water into teapot
Pour tea into cup
Turn off tap
Add tea bags to teapot
Fill kettle
Add milk
Turn on tap
Warm teapot
Heat kettle

Step 3 - Determine the order of steps needed to solve the problem

```
Turn on tap
Fill kettle
Turn off tap
Heat kettle
Warm teapot
Add teabags to teapot
Pour water into teapot
Pour tea into cup
Add milk
```

Step 4 - Identify any sub procedures

The process of filling and heating the water in a kettle is a procedure that could be considered a subroutine. It is a procedure that is called on in a number of situations and can be effectively reused. Designing methods like this that can be reused in a variety of circumstances can speed up software development and reduce testing time as the method is already tested and working and can simply be "plugged in" to the required solution.

We could make a sub procedure that we can plug in:

```
Procedure boilKettle ()
Turn on tap
Fill kettle
Turn off tap
Turn on kettle
```

And the final solution would be:

```
boilKettle ()
Warm teapot
Add teabags to teapot
Pour water into teapot
Pour tea into cup
Add milk
```

Notice that many of these procedures can be broken down into further sub-procedures. The level of decomposition will depend on the type of program required. The design of a device driver for a piece of hardware for example may require some low level programming. This level of decomposition may lead to further and more complex logical thinking which is covered below.

4.1.4-4.1.8 Thinking logically

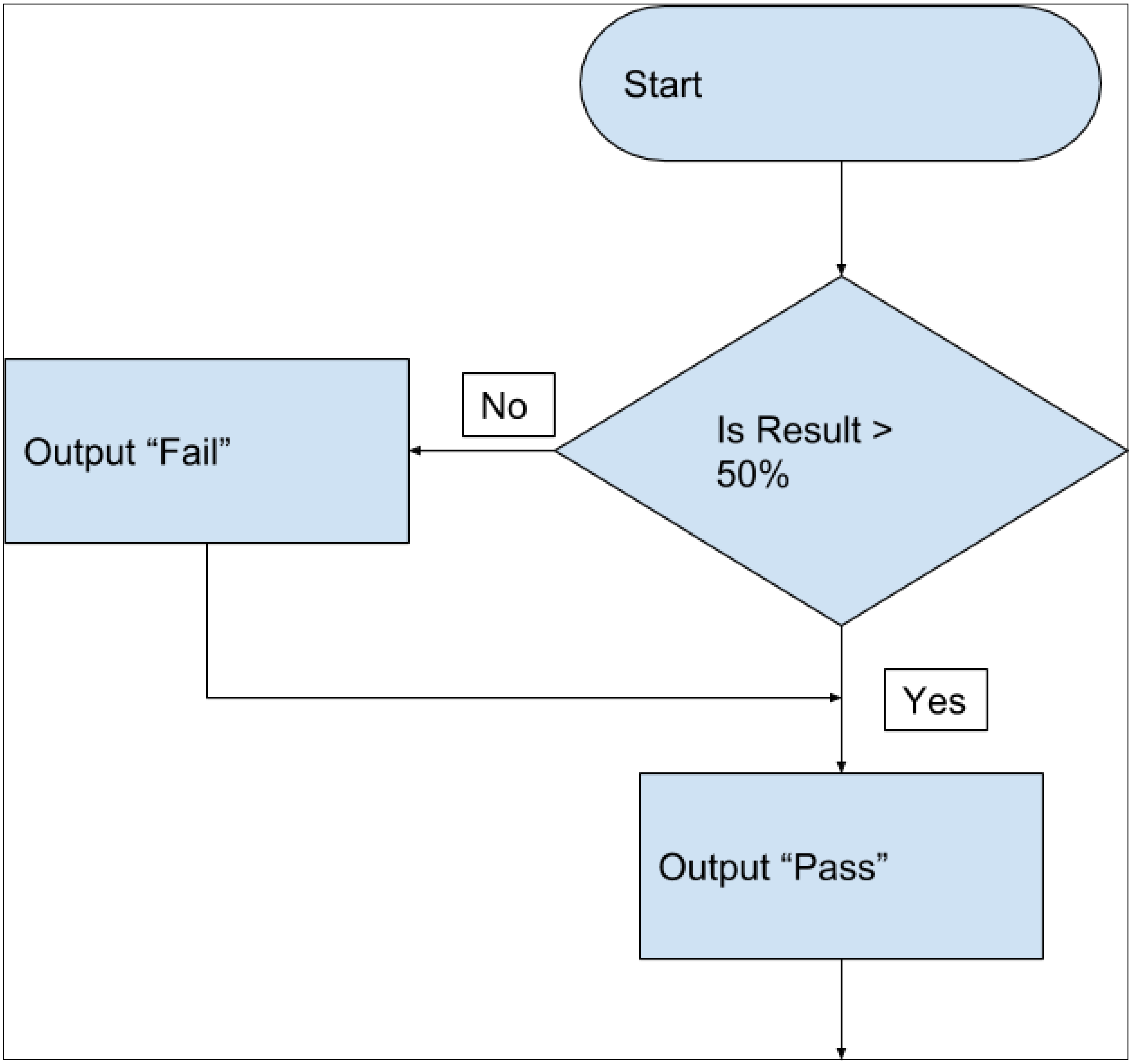
This is one of the fundamental parts of computational thinking. Computers process data logically but this is not "thinking". In order for the computer to work it needs to be given instructions. In order for these instructions to work then the programmer must think logically in order to design the program. Logical thinking is also fun and there are a great many ways for you to develop your logical thinking skills such as Suduko and Kakuro or Lightbot.

4.1.4 Identify when decision-making is required in a specified situation.

4.1.5 Identify the decisions required for the solution to a specified problem.

In many cases the steps required to solve a problem such as the example given above require no decision making and are linear solutions. However if we may have simple situations that require a decision to be made and there are two alternative decisions to be made.

EG A system that automatically grades exam papers. Part of the program will involve a decision. This can be expressed using a flowchart:



Or alternatively in pseudocode:

```
if GRADE>50 then
Output "Pass"
else
Output "Fail"
Endif
```

In this case the decisions that need to be identified in order to solve the problem is the logical condition determining if the exam is a pass or fail.

4.1.6 Identify the condition associated with a given decision in a specified problem.

Boolean logic

Boolean logic is named after George Boole, a famous 19th Century mathematician. In this logic all values are expressed as either true or false. This fits well with binary systems which can be expressed as either 0 or 1. Boolean operators exist that can be used to manipulate the basic true false values. Examples of operators are as follows

Operator	Meaning
=	Is equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
≠	Not equal to
AND	The output is true if both inputs are true
OR	The output is true if one of the inputs is true
NOT	The output is true if the input is false
NAND	The output is true if either or both inputs are false.
NOR	The output is true if both inputs are false
XOR	The output is true if only one of the inputs is true and only one is false. Known as exclusive OR.

These may be expressed in a logic table where 0 is FALSE and 1 is TRUE.

AND

Input A	Input B	Output Z
0	0	0
0	1	0
1	0	0
1	1	1

OR

Input A	Input B	Output Z
0	0	0
0	1	1
1	0	1
1	1	1

NOT

Input A	Output Z
0	1
1	0

NAND

Input A	Input B	Output Z
0	0	1
0	1	1
1	0	1
1	1	0

NOR

Input A	Input B	Output Z
0	0	1
0	1	0
1	0	0
1	1	0

XOR

Input A	Input B	Output Z
0	0	0
0	1	1
1	0	1
1	1	0

These can be used to create logic table to find the solution to problems. For example a student is only allowed to play computer games if their homework is done and their room is tidy. The solution to this can be illustrated in a logic table as follows:

Homework done	Room tidy	Output Z
0	0	0
0	1	0
1	0	0
1	1	1

More details of how to solve logic problems is covered in Section 2.1.11.

Iteration and testing conditions

Iteration in computing is the repetition of a block of statements within a computer program. In practice this repetition can take place a set number of times or can repeat based on a set of logical criteria that change over time.

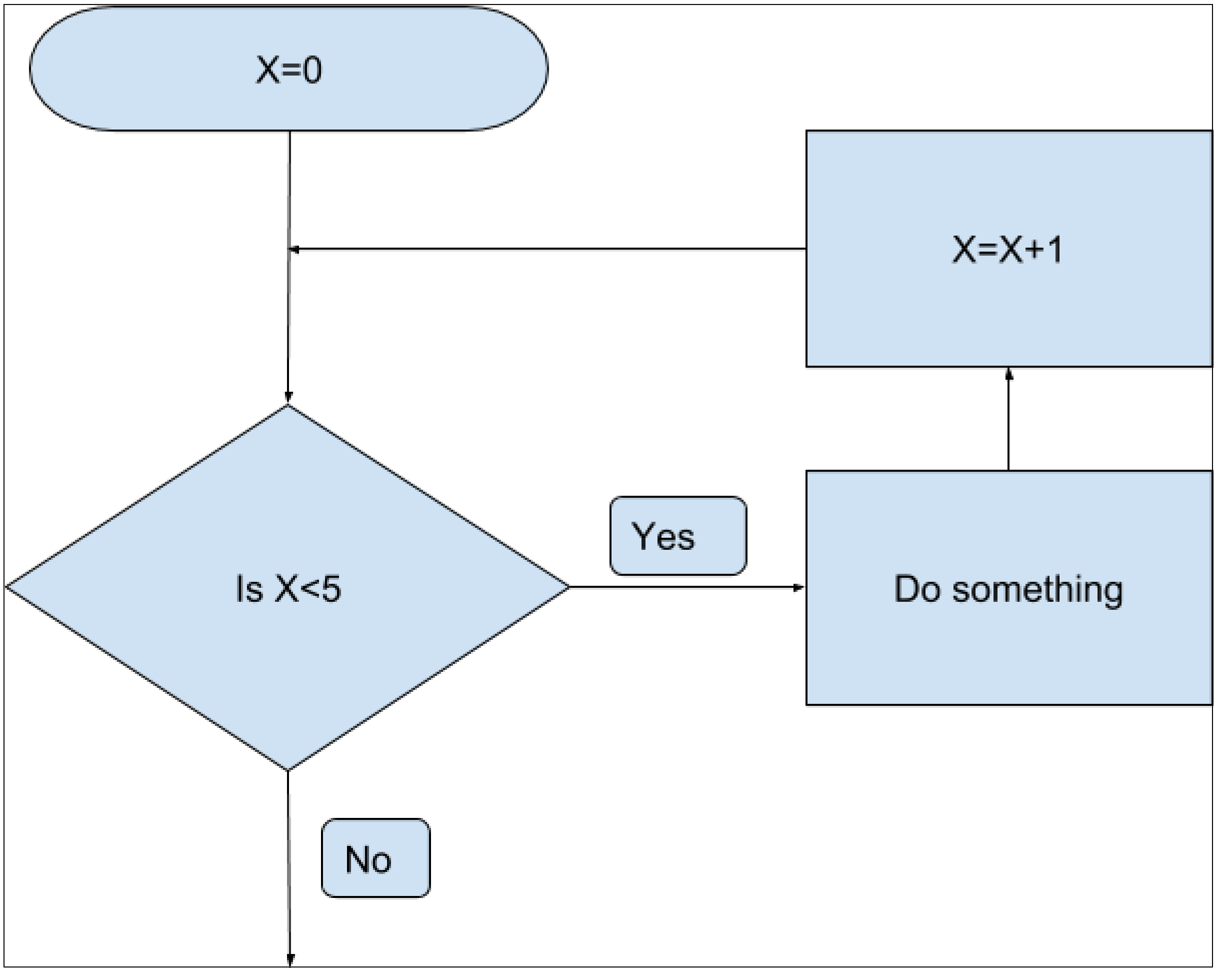
There are two main ways this is achieved in programming which are a for loop and a while loop.

For loop

In the IB pseudocode document the syntax for a *for* loop is a *from to* loop. In most programming languages the syntax of a for loop is:

```
for (INITIALIZATION; CONDITION; AFTERTHOUGHT)
{
   // Code for the for-loop's body goes here.
}
```

This may be expressed in a flowchart as:



In JAVA the syntax would be

```
For (int x= 0; x<5;x++){
//do something
}
```

In IB psueudocode it would be as follows:

```
loop X from 0 to 4
//do something
end loop
```

While loop

A while loop is an iteration but in this case the loop must repeat until a certain condition is met. This can be programmed in JAVA in the following way

```
while (COUNT<5) {
COUNT = COUNT + 1
}
```

In IB pseudocode this would look like this:

```
loop while COUNT<5
COUNT = COUNT +1
end loop
```

A variation of the while loop is the do while loop in which a condition is first run before the check is made. The difference is that the evaluation of the expression is done at the end of the loop:

```
do {
//Statements
}
while (condition)
```

4.1.7 Explain the relationship between the decisions and conditions of a system.

In computing conditional statements are features which perform different computations depending on whether a boolean expression evaluates as true or false. The basic structure of this statements using IB pseudocode is as follows:

```
if (condition) then
//Do something
else
//Do something else
end if
```

This can be used in a number of contexts in computer programming.

4.1.8 Deduce logical rules for real-world situations.

Thinking ahead

Thinking concurrently

Thinking abstractly

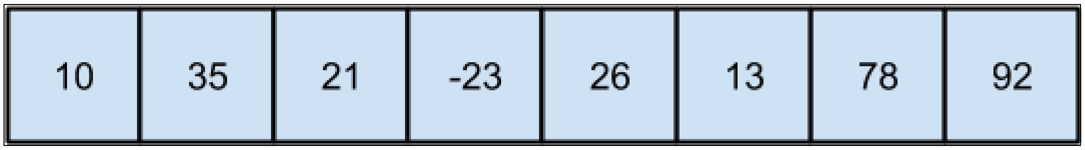
4.2 Connecting computational thinking and program design

4.2.1 Describe the characteristics of standard algorithms on linear arrays.

Sequential search

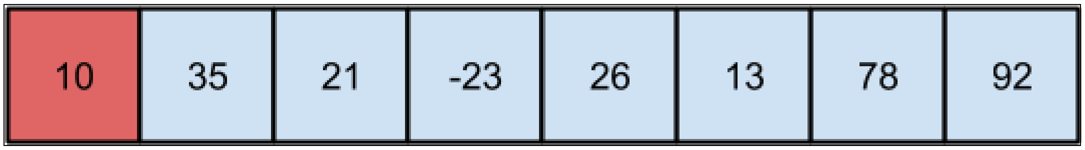
In computer science, linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

For example consider the following array of integers:



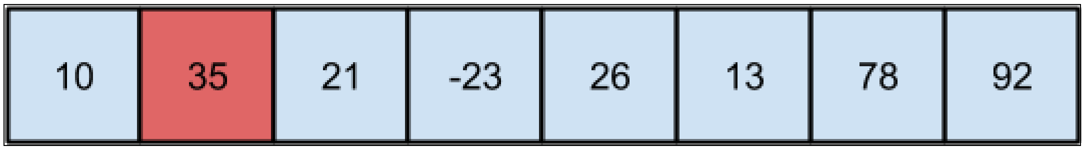
10	35	21	-23	26	13	78	92

Let us imagine we are searching for the number 21. First go to the first element of the array.



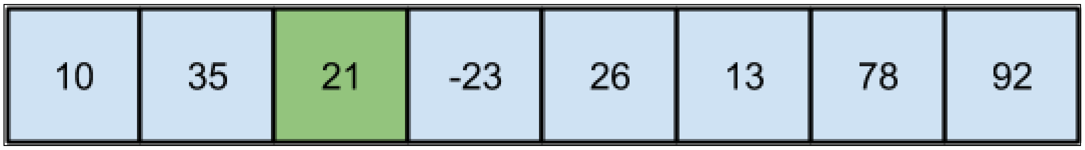
🟥 10	35	21	-23	26	13	78	92

Is it equal to 21? No. Then move to the next element.



10	🟥 35	21	-23	26	13	78	92

Is it equal to 21? No. Then move to the next element.



10	35	🟩 21	-23	26	13	78	92

Is it equal to 21? Yes.
Here is an example of a linear search in pseudocode:

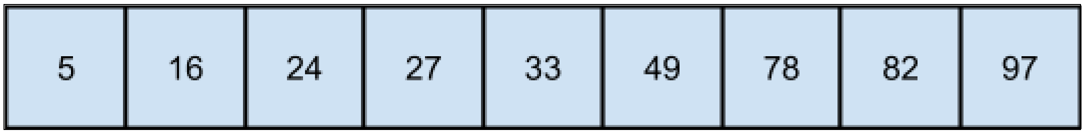
```
N=0
loop while ARRAY.hasNext()
if ARRAY[N] == SEARCHTERM then
output N
end if
N=N+1
end loop
output -1
```

The method returns the position of the search term if it is found otherwise it returns a -1.

Binary Search

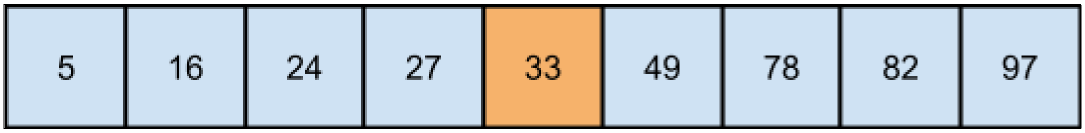
In computer science, a binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

For example consider we are searching the following sorted array of integers for the number 27.



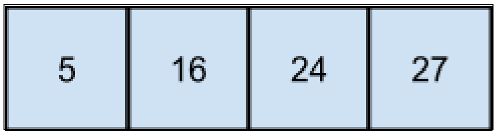
5	16	24	27	33	49	78	82	97

First find the middle position



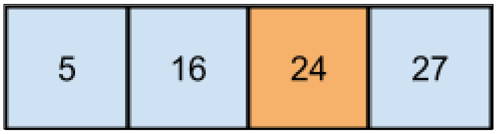
5	16	24	27	33	49	78	82	97

Is 16<33? Yes. Discard the right hand half.



5	16	24	27

Next find the middle position



5	16	24	27

Is 16<24. Yes. Discard the right hand half.

27

Next find the middle position

27

Is 27<27? No
Is 27>27? No

Then we have found the term.

There are several variations of this method but it can be written in an iterative way as (IB sample algorithms):

```
public int binarySearchA(int target, int[] nums, int size)
   {
      //  An iterative binary search.  Size = size of num array.
      //  If found, returns position.  Else returns -1.
      int middle, low, high;
      boolean found = false;
      low = 0;
      high = size-1;
      middle = -1;
      while (high >= low && !found)
      {  middle = (low + high) / 2;
         if (target < nums[middle])
         {  high = middle - 1; }
         else if (target > nums[middle])
         {  low = middle + 1;  }
         else
         {  found = true;  }
      }
      if (found)
      {  return middle; }
      else
      {  return -1;  }
   }
```

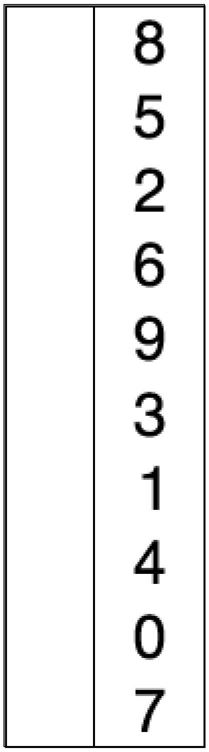
Bubble Sort

Bubble sort, sometimes incorrectly referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.

There are several variations of this method but it can be written in an iterative way as (IB sample algorithms):

```
public void bubbleSortB(int[] nums, int size)
   {
      //  An ascending bubble sort, but this does not perform a
      //  specific number of passes.  Instead, it stops if an entire pass
      //  completes WITHOUT any swaps occurring.
      int current, temp;
      boolean done;
      do
      {
         done = true ;
         for(current = 0; current < size-1 ; current = current + 1)
         {  if (nums[current] > nums[current + 1])
            {
               temp = nums[current];
               nums[current] = nums[current+1];
               nums[current+1] = temp;
               done = false;
            }
         }
      } while (!done);
   }
```

Selection sort.



	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

In computer science, selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(*n*2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

There are several variations of this method but it can be written in an iterative way as (IB sample algorithms):

```
public void selectionSortB(int[] nums,int size)
   {
      int first, current, least, temp;
      for(first = 0; first < size; first = first + 1)
      {
         least = first;
         for(current = first+1; current < size; current = current + 1)
         {  if (nums[current] < nums[least])
            {  least = current; }
         }
         temp = nums[least];
         nums[least] = nums[first];
         nums[first] = temp;
      }
   }
```

4.2.2 Outline the standard operations of collections.
Addition and retrieval of data.

4.2.3 Discuss an algorithm to solve a specific problem.

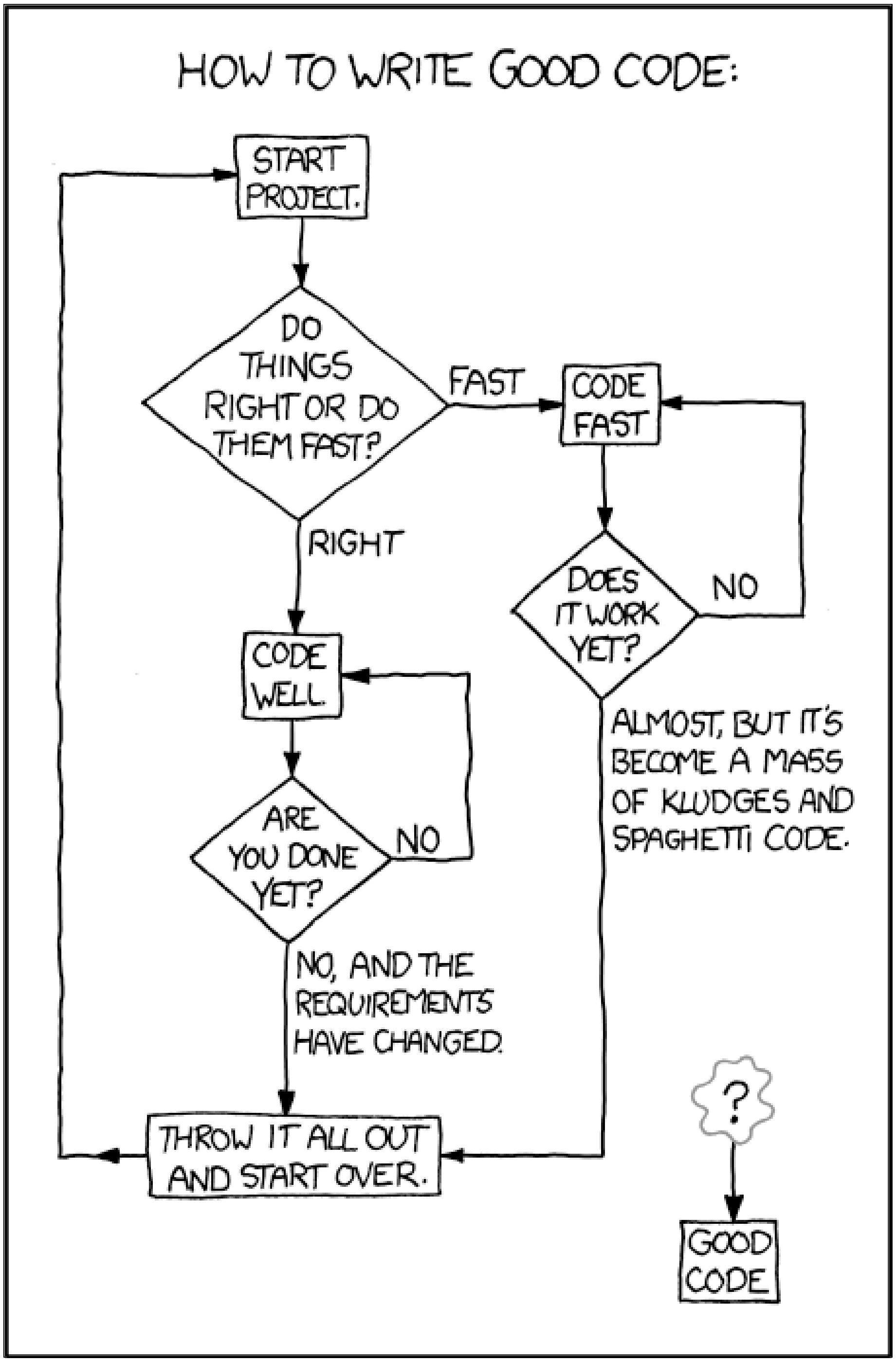
EG Linear search vs binary search

Binary search requires the data to be sorted linear search does not.
Comparing the two we find that the efficiency of the linear search is O(n) whereas the efficiency of the binary search is O($\log_2$n).

Array Size	Efficiency	
n	Linear Search O(n)	Binary Search O(logn)
10	10	3
100	100	7
1000	1000	10
10000	10000	13

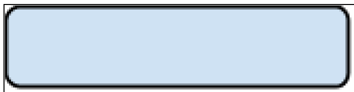
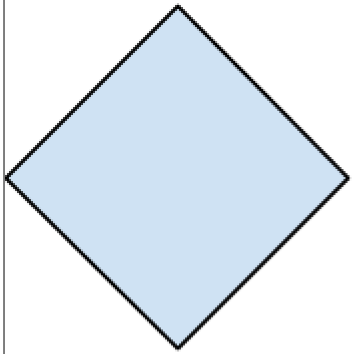
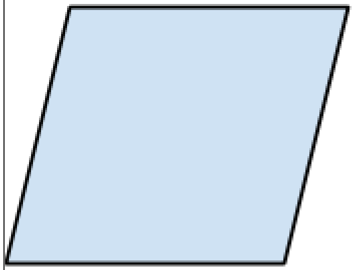
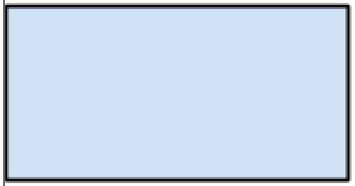
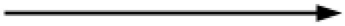
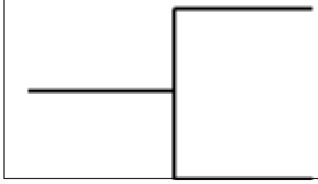
Binary search requires random access to the data whereas linear search does not. Because of this binary search will not work on linked lists for example. Linear search only requires sequential access.

4.2.3 Analyse an algorithm presented as a flow chart.


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

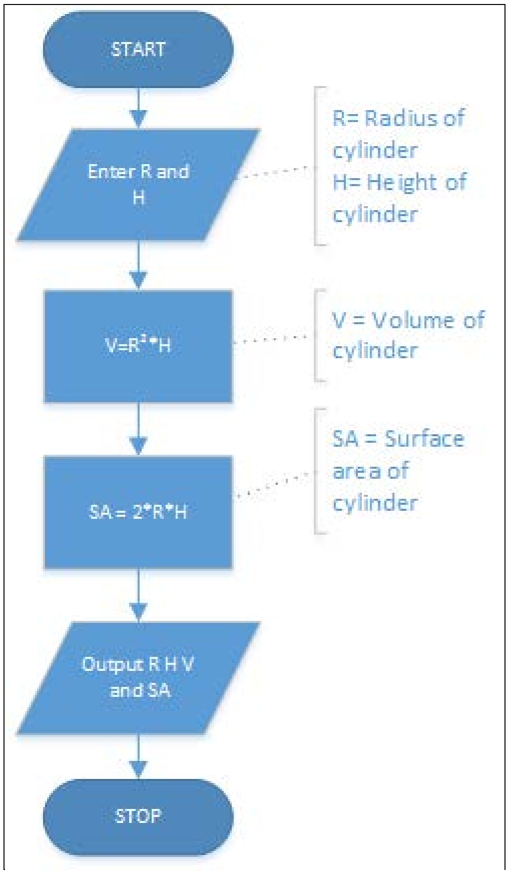
An algorithm is a detailed sequence of steps that are needed to solve a problem. A flowchart is a graphical representation of an algorithm.

Basic flowchart symbols

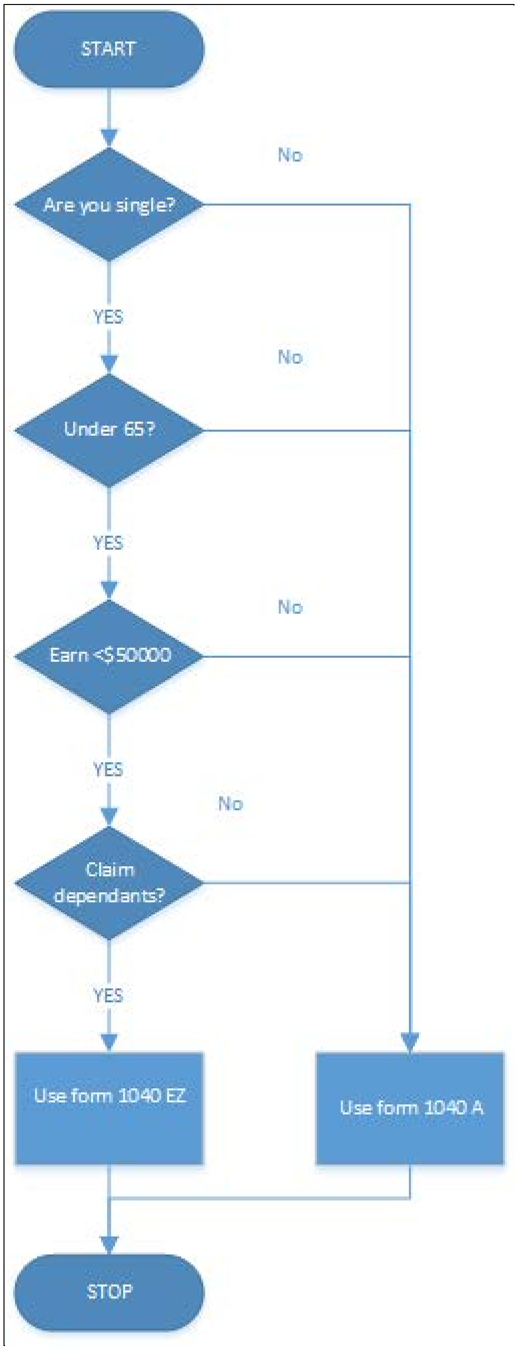
	Start/Stop.
	Question/Decision used in branching (eg IF/THEN/ELSE).
	Input or an output operation (eg Print).
	Process to be carried out (eg addition).
	Direction of logic flow in a program.
	Annotation

Flowchart examples

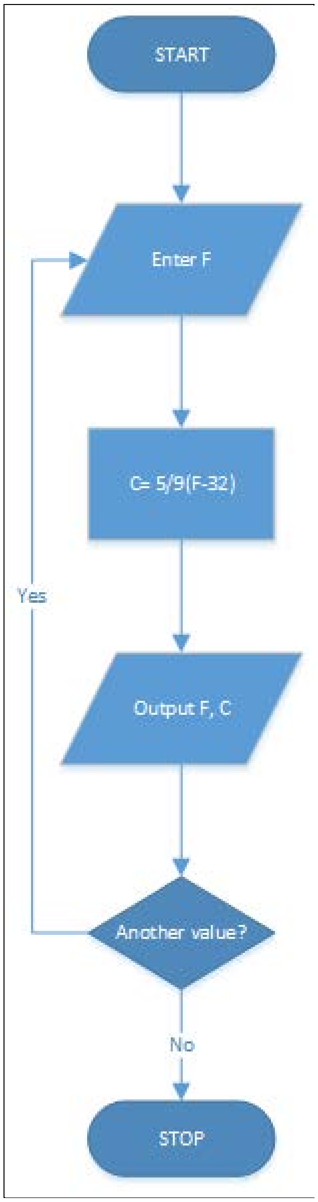
Sequential structure



Selection/branching structure



Repetition/looping structure



4.2.3 Analyse an algorithm presented as pseudocode.
Make sure that you have read the two documents from the IB “Pseudocode in examinations” and “Approved notation for developing pseudocode”. See assignments on e-learning.

4.2.4 Construct pseudocode to represent an algorithm.
Make sure that you have read the two documents from the IB “Pseudocode in examinations” and “Approved notation for developing pseudocode”. See assignments on e-learning.

4.2.5 Suggest suitable algorithms to solve a specific problem.
See assignments on e-learning.

4.2.6 Deduce the efficiency of an algorithm in the context of its use.

Lesson 39….. Complexity Analysis (Big O). See assignments on e-learning.

4.2.7 Determine the number of times a step in an algorithm will be performed for given input data.

See assignments on e-learning.

4.3 introduction to programming

Nature of programming languages

4.3.1 State the fundamental operations of a computer

A fundamental operation could be something like, multiply two numbers, store a number, move a number to another location etc. These are operations that do not require the processor to go through a large number of sub operations to reach a result.

4.3.2 Distinguish between fundamental and compound operations of a computer

For example, find the largest is a compound operation.
A compound operation is an operation that involves a number of stages/other operations. Think of it as a group of operations that combine together to form an operation.

4.3.3 Explain the essential features of a computer language

Languages in General Terms

- All languages are subject to a set of rules governing the set of valid characters that can be used
    - word construction
    - sentence construction
- The rules of sentence construction are called syntax
- To use a language correctly you need to know about these and
    - vocabulary - the set of valid words
    - semantics - ensuring that what is written makes sense
- You need to be able to recognize language 'tokens'

Syntax rules

- English language descriptions of language rules can be very cumbersome, and lead to ambiguity
- Syntax diagrams are a neat way of expressing language rules
    - they should be completely unambiguous
    - they usually involve other syntactic elements (tokens)
- What would a syntax diagram look like for
    - a word
    - a sentence
    - prose

Syntax vs Semantics

- Syntax - the grammatical arrangement of words to show their connection and relation; the set of rules governing this arrangement.
- Semantics - relate to meaning in language

- A sentence in English may be syntactically correct
    - it obeys the language rules
- But not necessarily semantically correct
    - it has no valid meaning
- Both syntax and semantics are important

Programming language building blocks

- A computer program is like a set of instructions
- Each instruction is termed a statement
- Each program statement has a terminating character
- Translate into syntax diagrams

4.3.4 Explain the need for higher level languages

In computer science, a high-level programming language is a programming language with strong abstraction from the details of the computer. In comparison to low-level programming languages, it may use natural language elements, be easier to use, or may automate (or even hide entirely) significant areas of computing systems (e.g. memory management), making the process of developing a program simpler and more understandable relative to a lower-level language. The amount of abstraction provided defines how "high-level" a programming language is.
Examples of high-level programming languages include Java, Lisp, R, Python and Ruby.

4.3.5 Outline the need for a translation process from a higher level language to machine executable code

Interpreted

Interpreted languages are read and then executed directly, with no compilation stage. A program called an interpreter reads each program statement following the program flow, decides what to do, and does it. A hybrid of an interpreter and a compiler will compile the statement into machine code and execute that; the machine code is then discarded, to be interpreted anew if the line is executed again. Interpreters are commonly the simplest implementations, compared to the other two variants listed here.

Compiled

Compiled languages are transformed into an executable form before running. There are two types of compilation:

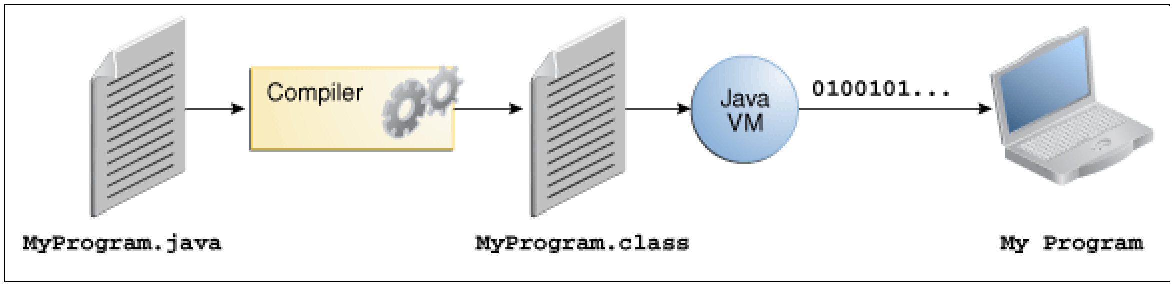
Machine code generation

Some compilers compile source code directly into machine code. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages. See assembly language.

Scanning
Parsing
Optimise
Code generation

Intermediate representations

When a language is compiled to an intermediate representation, that representation can be optimized or saved for later execution without the need to re-read the source file. When the intermediate representation is saved, it is often represented as byte code. The intermediate representation must then be interpreted or further compiled to execute it. Virtual machines that execute bytecode directly or transform it further into machine code have blurred the once clear distinction between intermediate representations and truly compiled languages.



Java uses a two-step compilation process. Java source code is compiled down to "bytecode" by the Java compiler. The bytecode is executed by Java Virtual Machine (JVM). The current version of Sun HotSpot JVM uses a technique called Just-in-time (JIT) compilation to compile the bytecode to the native instructions understood by the CPU on the fly at run time.

Use of programming languages

4.3.6 Define the terms: variable, constant, operator, object.

Variable

In computing a variable relates to a name or identifier which relates to a value at a particular memory storage location.

Constant

In computing a constant is an identifier whose associated value cannot typically be changed by the program during runtime as opposed to a variable whose value can be altered.

Operator

Programming languages support a range of operators which behave like functions. Operators are special symbols that perform specific operations on operands, and then return a result. For example operators from JAVA can be seen below:

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

4.3.7 Define the operators =, ≠, <, <=, >, >=, mod, div.

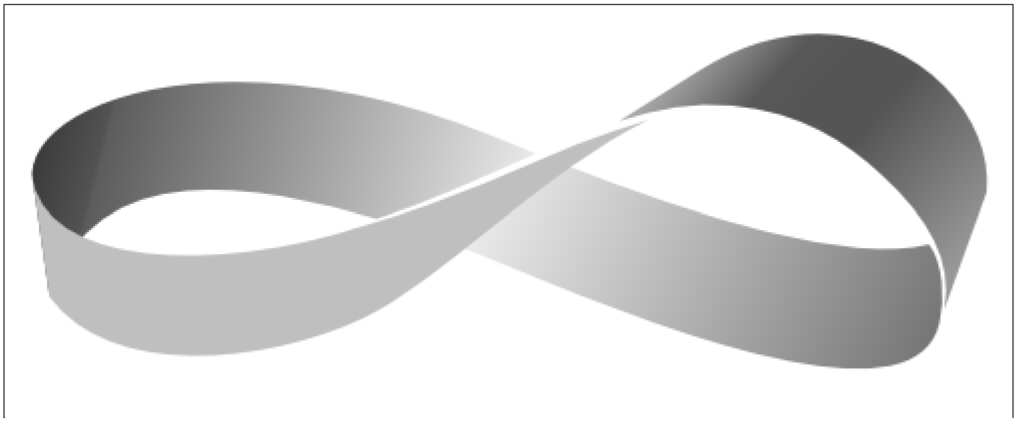
Operator	Definition
=	Assignment operator eg `int age = 18` This assigns the value of 18 to the variable age
≠	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

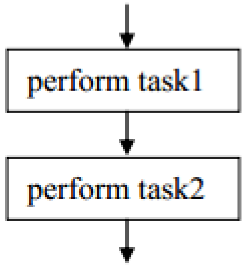
mod	Modulus operator which will give a remainder of a division. .
	eg 5 mod 2 = 1
	In JAVA the operator is %
	eg to find if a number is odd or even the following code can be used:
	`boolean isEven`
	`if ( (a % 2) == 0)`
	`{`
	`    isEven = true`
	`}`
	`else`
	`{`
	`    isEven = false`
	`}`

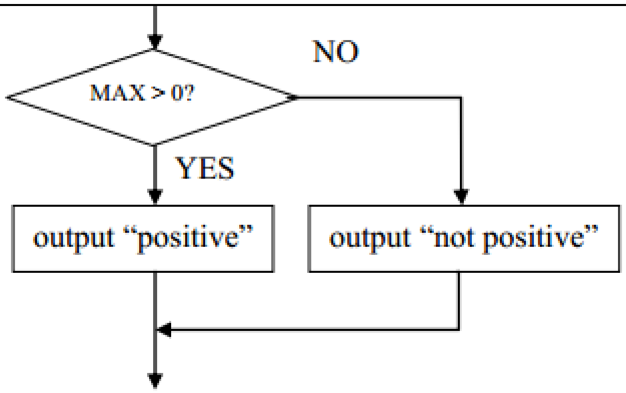
div	Integer part of quotient. eg 15 div 7 = 2

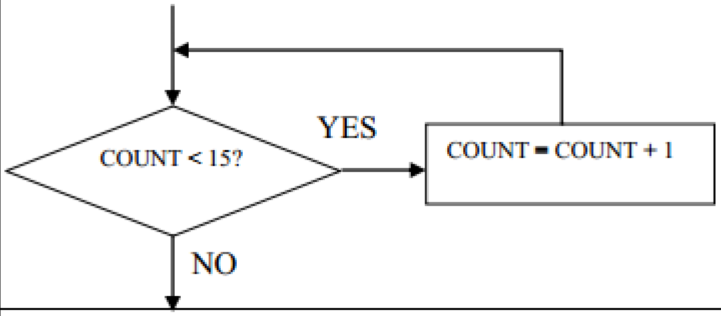
4.3.8 Analyse the use of variables, constants and operators in algorithms.

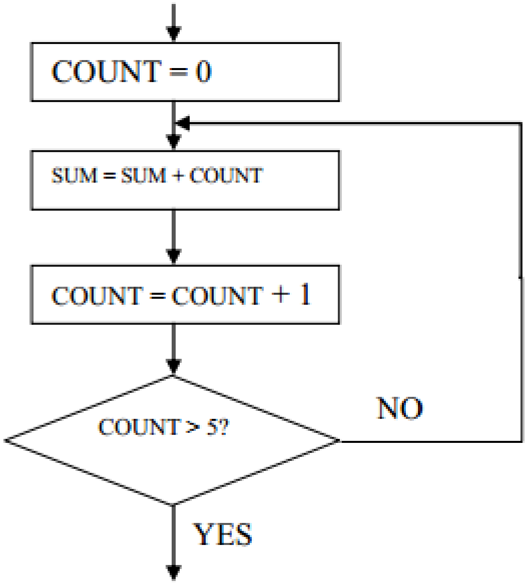
4.3.9 Construct algorithms using loops, branching.



Operation	Flowchart example	Pseudocode example
sequential operations		perform task1 perform task2

conditional operations	 NO
	MAX > 0?
	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

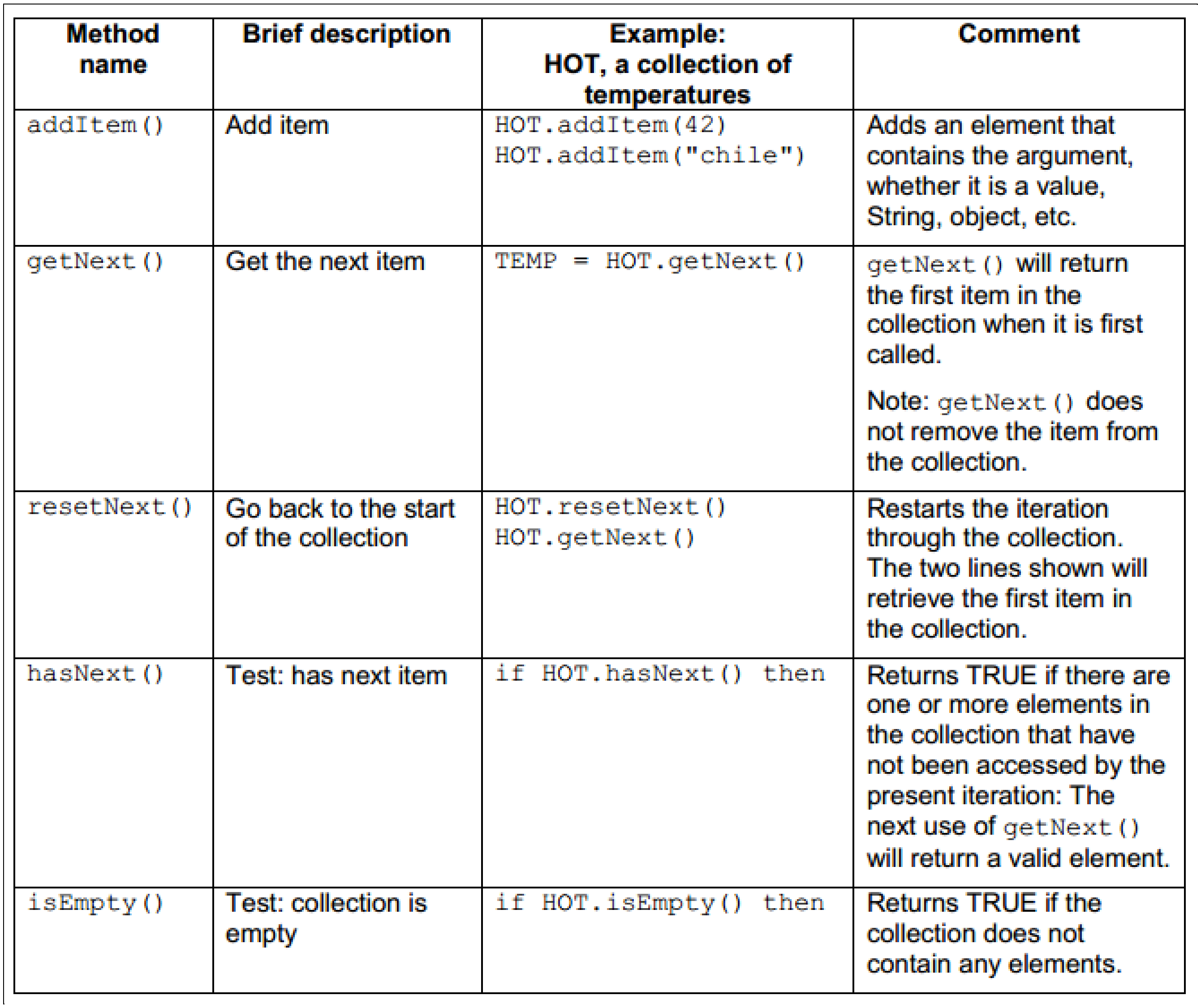
4.3.10 Describe the characteristics and applications of a collection.

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
```

4.3.11 Construct algorithms using the access methods of a collection.



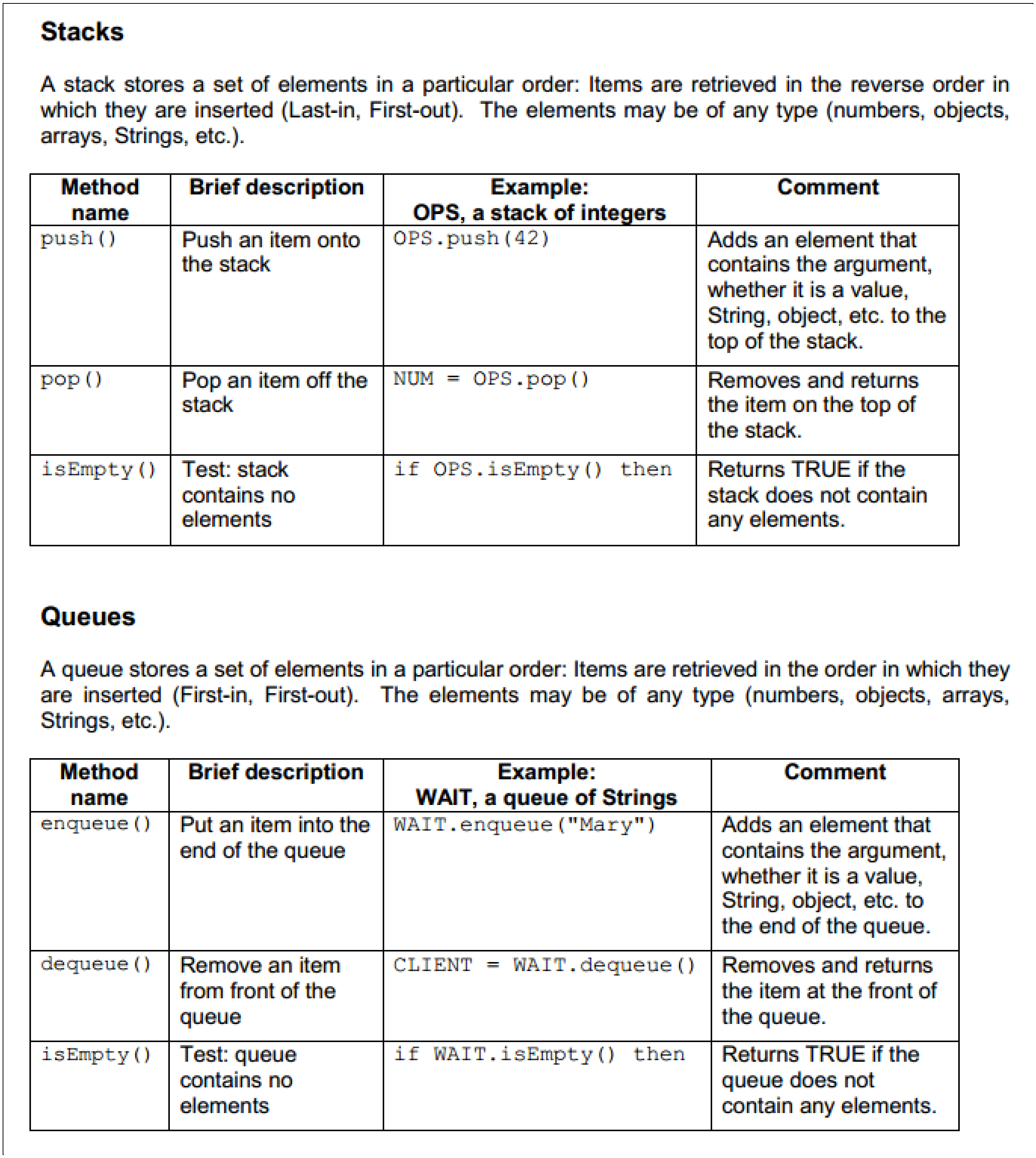
Method name	Brief description	Example: HOT, a collection of temperatures	Comment

`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.



Stacks

A stack stores a set of elements in a particular order: Items are retrieved in the reverse order in which they are inserted (Last-in, First-out). The elements may be of any type (numbers, objects, arrays, Strings, etc.).

Method name	Brief description	Example: OPS, a stack of integers	Comment
`push()`	Push an item onto the stack	`OPS.push(42)`	Adds an element that contains the argument, whether it is a value, String, object, etc. to the top of the stack.
`pop()`	Pop an item off the stack	`NUM = OPS.pop()`	Removes and returns the item on the top of the stack.
`isEmpty()`	Test: stack contains no elements	`if OPS.isEmpty() then`	Returns TRUE if the stack does not contain any elements.

Queues

A queue stores a set of elements in a particular order: Items are retrieved in the order in which they are inserted (First-in, First-out). The elements may be of any type (numbers, objects, arrays, Strings, etc.).

Method name	Brief description	Example: WAIT, a queue of Strings	Comment
`enqueue()`	Put an item into the end of the queue	`WAIT.enqueue("Mary")`	Adds an element that contains the argument, whether it is a value, String, object, etc. to the end of the queue.
`dequeue()`	Remove an item from front of the queue	`CLIENT = WAIT.dequeue()`	Removes and returns the item at the front of the queue.
`isEmpty()`	Test: queue contains no elements	`if WAIT.isEmpty() then`	Returns TRUE if the queue does not contain any elements.

4.3.12 Discuss the need for sub-programmes and collections within programmed solutions.

4.3.13 Construct algorithms using pre-defined sub-programmes, one-dimensional arrays and/or collections.

Links

http://www.google.com/edu/computational-thinking/what-is-ct.html
http://www.cs4fn.org/computationalthinking/
http://csta.acm.org/Resources/sub/ResourceFiles/CompThinking.pdf

Now test yourself

Now test yourself!!

Revise terms: **https://quizlet.com/vn/545210775/**

Booklet Test: https://quizlet.com/vn/545211141