IB Computer Science Revision Notes

Topic 4 - Computational thinking, problem-solving and programming

4.1 General principles

Thinking procedurally

4.1.1 Identify the procedure appropriate to solving a problem

4.1.2 Evaluate whether the order in which activities are undertaken will result in the required outcome

4.1.3 Explain the role of sub-procedures in solving a problem

Sub-procedures are functions that have no use on their own but are used in other procedures to contribute to solving a larger problem, for example a function that searches an array for a given item in a library management system. Putting these tasks down into sub procedures has many advantages:

- The complexity of the main procedure is reduced (only the abstraction of the sub-procedure, the calling command, is displayed)
- This allows easier understanding of the working of the procedure
- Each sub-procedure has its own task
- If many procedures use the sub-procedure, code duplication can be avoided → reduced risk of bugs in the code
- Easier management of the code as each function can be developed on their own

Thinking logically

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

Every time there are two possible ways for the algorithm to go. For example, decision making is required to decide if the entered data is valid in a system.

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

Lets take the example that a system needs the age of the users. To prevent invalid values to be entered, a validation algorithm is used. The decisions required for this algorithm:

- Is a negative value entered?
- Is the value not greater than 150?
- Depending on the system, is the age below 18?

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

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

The conditions of a system determine the decisions that are made in the system. Because there are many decisions in the system, one decision can alter the conditions for an other.

4.1.8 Deduce logical rules for real world situations

Thinking ahead

4.1.9 Identify the inputs and outputs required in a solution

4.1.10 Identify pre-planning in a suggested problem and solution

4.1.11 Explain the need for pre-conditions when executing an algorithm

Algorithms make decisions based on some pre-conditions. So, they need some initial conditions after which they direct their working. Pre-conditions are often given as parameters to the algorithm.

4.1.12 Outline the pre- and post-conditions to a specified problem

4.1.13 Identify exceptions that need to be considered in a specific problem solution

Thinking concurrently

4.1.14 Identify the parts of a solution that could be implemented concurrently

Concurrent computing is a form of computing in which several computations are executing during overlapping time periods – concurrently – instead of sequentially (one completing before the next starts) Concurrent computing [https://en.wikipedia.org/wiki/Concurrent computing].

Many processes can be implemented concurrently:

- Some calculations (Brute force calculations are usually implemented concurrently)
- GUI is usually implemented concurrently
- The programs in a modern computer run concurrently (parallel)

4.1.15 Describe how concurrent processing can be used to solve a problem

In a production line many processes are running parallel. Concurrent processing plays a crucial role here as it is what allows the system to execute a large amount of tasks simultaneously.

4.1.16 Evaluate the decision to use concurrent processing in solving a problem

You need to think ahead. Before you start designing the solution try to think on which tasks can be run independently from each other. For example, in a calendar software, the periodical check if any reminders are due can happen independent from user input, so can run parallel to the rest of the solution.

Thinking abstractly

4.1.17 Identify examples of abstraction

An abstraction could be a car engine. You don't need to know how it works to be able to operate it, you only need to know how its abstraction works. So, creating an abstraction requires a lots of thinking ahead as you need to select the pieces of information that are relevant for an abstraction.

4.1.18 Explain why abstraction is required in the derivation of computational solutions for a specified situation

Computers cannot store an infinite number of things so we need to reduce them to abstractions, containing only necessary information. Also, this properties can then be grouped together into objects, which are a fundamental concept in computing and algorithms.

4.1.19 Construct an abstraction from a specified situation

Lets say we need to create an abstraction from the people in a school. We can decompose the people in the school into teachers and students. Students can aslo be decomposed into boys and girls.

4.1.20 Distinguish between a real-world entity and its abstraction

A car engine. It is made up of hundreds of parts, but we still need no nowledge on how they work. We only need to know how its abstraction - the pedals, gearshifting and the steering - work to be able to run it. In fact, the similarity of the abstractions allow us to be able to operate many different kinds of engines and vehicles, regardless on what type of engine they are using.

4.2 Connecting computational thinking and program design

4.2.1 Describe the characteristics of standard algorithms on linear arrays

Sequential search: The algorithm loops through the array one-by-one until the required item is found or the end of the array has been reached. It is the simplest of all search algorithms.

Efficiency for a list consisting of n elements: O(n)

Binary search: The algorithm needs an ordered list to work. It starts in the middle of the array and checks if the desired result is either in the first or second half of the list. The other half is then discarded and the process is used for this sub-array until there is only one element left.

Efficiency for a list consisting of n elements: O(log$n$). However, if the list is unsorted, the algorithm is useless.

Bubble sort: This is an algorithm to sort an unsorted array, based on comparing and switching items.

6 5 3 1 8 7 2 4

Efficiency for a list consisting of n elements: $O(n^2)$

Selection sort: The algorithm starts with the first element in the array. Then it loops through the rest of the array and finds the smallest element and switches it with the first one. Then it goes on to the second element and repeats the procedure until it reaches the end of the array.

Example: an array with content 4|2|1|6|3|5 should be sorted:

4	2	*1*	6	3	5	Minimum is 1, so switch first and third element
1	*2*	4	6	3	5	Minimum is 2. Because it is already at the first position nothing is changed
1	**2**	*4*	6	*3*	5	Switch 4 and 3
1	**2**	**3**	*6*	*4*	5	Switch 6 and 4
1	**2**	**3**	**4**	*6*	*5*	Switch 6 and 5
1	**2**	**3**	**4**	**5**	**6**	The array is now sorted

Efficiency for a list consisting of n elements: $O(n^2)$

4.2.2 Outline the standard operations of collections

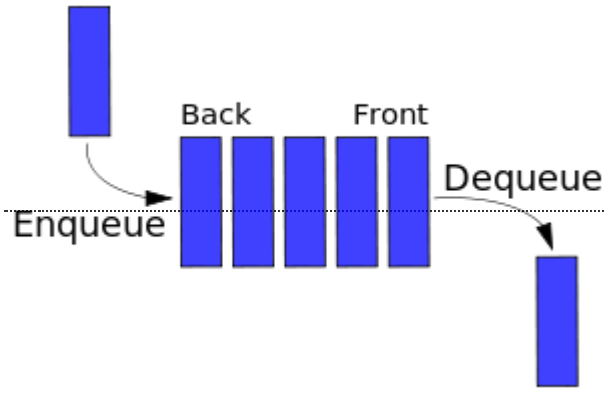
Lists: A finite ordered collection of values, where the same value may occur more than once.

Operations:

- constructor for creating an empty list
- testing whether or not a list is empty
- prepending an entity to a list
- appending an entity to a list
- determining the first component ("head") of the list
- referring to the list consisting of all components except for the first one (this is the "tail" of the list)

Stack: A special kind of collection where the only operations are the addition or removal of an entity to the collection, known as *push* and *pop*. This makes a stack a Last-In-First-Out (LIFO) data structure.

Queue: A special kind of collection where the entities are kept in order and the only operations permitted are the addition or removal on an entity, where the first added entity is removed first. This makes a queue a First-In-First-Out (FIFO) data structure.



4.2.3 Discuss an algorithm to solve a specific problem

We are expected to discuss the differences between algorithms used to solve similar problems, for example if binary search is better or not than sequential search for a collection.

4.2.4 Analyse an algorithm presented as a flow chart

Syllabus link:
Examination questions may involve variables, calculations, simple and nested loops, simple conditionals and multiple or nested conditionals.

This would include tracing an algorithm as well as assessing its correctness.

Students will not be expected to construct a flow chart to represent an algorithm in an externally assessed component.

4.2.5 Analyse an algorithm presented as pseudocode

Syllabus link:
Examination questions may involve variables, calculations, simple and nested loops, simple conditionals and multiple or nested conditionals.

This would include tracing an algorithm as well as assessing its correctness.

4.2.6 Construct pseudocode to represent an algorithm

We are expected to construct pseudocode for some simple operations like checking a mail address for correctness.

Pseudocode notation approved by IB

4.2.7 Suggest suitable algorithms to solve a specific problem

Syllabus link:
Suitable algorithms may include both standard algorithms and novel algorithms. Suitable may include considerations of efficiency, correctness, reliability and flexibility. Students are expected to suggest algorithms that will actually solve the problem successfully.

4.2.8 Deduce the efficiency of an algorithm in the context of its use

We should be aware how different techniques like nested loops can influence the efficiency of an algorithm (a nested loop decreses efficiency to $O(n^2)$)

We also should be able to suggest changes in an algorithm to increase efficiency, for example setting a flag to stop search immediately after an item is found.

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

Questions will involve specific algorithms in pseudocode or flow charts and we are expected to give an acual number of steps required for a given situation. This involves that we have to trace the algorithm.

4.3 Introduction to programming

Nature of programming languages

4.3.1 State the fundamental operations of a computer

Add, compare, retrieve and store data.

All of the more complex operations compose of a combination of these simple operations

4.3.2 Distinguish between fundamental and compound operations of a computer

Examples of fundamental operations:

- Add one number to another
- Store a number in memory
- Compare two numbers to find which one is larger

Examples of compound operations of a computer:

- Find the largest out of several numbers
- Find the modulus of a division

4.3.3 Explaine the essential features of a computer language

- Fixed vocabulary
- Commands to store, retrieve and modify data
- Defines operations a computer can execute

4.3.4 Explain the need for higher level languages

It is very hard and prone to errors to write programs in machine language. A high level language looks more like english and is thus more readable for programmers. This allows them to focus more on the problem solving itself than on translating their solution into machine-executable code. Also, because every different machine has its own machine instruction set, it is easier to translate programs written in high level languages for many machines than machine code.

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

Higher level languages are easily readable for humans but computers ultimately understand one form of instructions - machine code. So, programs written in high level languages need to be translated into machine language so that computers can execute them.

Use of programming languages

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

Variable: is an entity in a computer program that refers to a location in primary memory that holds a value. There are different *variable types*, like integers and strings but they only differ in the way the information in memory is interpreted.

Constant: is a special type of variable those content cannot be changed once it is initialised. It is used to store values in a program that do not need to be changed throughout the runtime, for example a mathematics program would store π in a constant as its value does not need to be changed.

Operator: +, -, =, *, /, +=, -=. They are symbols of infix operations that determine what operation should be carried out on the parameter values or how to compare values in a boolean operation.

Object: is an abstract entity in object oriented programming. It tries to model a real-world thing by bundling its properties into one entity - the object - so that they can be treated together. It also simplifies programming as many variables belonging to one object can be grouped together in a logical way.

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

=: This operator assigns a value to a variable. In boolean operations it checks if two values are the same or not.

≠: This comparative operator compares if two values are the same or not. It is used in boolean operations.

<, >: This comparative operators check if the values are smaller or greater than each other.

⇐, >=: This comparative operators check if two values are smaller, greater or equal than each other.

mod: This infix operand returnes the modulus, the remainder of a division.

div: Whole number division. Any remainder is discarded.

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

A constant is used to store a value that is not supposed to change during the runtime on an algorithm, for example a constant storing π in an algorithm. A variable is used to store values that change when an algorithm is executed, like the result of an addition

4.3.9 Construct algorithms using loops, branching

Syllabus link:
Teachers must ensure algorithms use the symbols from the approved notation sheet Pseudocode notation approved by IB.

4.3.10 Describe the characteristics and applications of a collection

A collection is a grouping os some variable number of data items that have some shared significance to the problem being solved and the need to be operated upon together in some controlled fashion. Collection_(abstract_data_type) [https://en.wikipedia.org/wiki/Collection_(abstract_data_type)].

Collections may find applications in any type of programs that deal with many similar data items at once, for example a library program keeping track of books.

4.3.11 Construct algorithms using the access methods of a collection

Be sure that you use the specific operations of specific kinds of collections (pou and push for stacks, for example)

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

Sorting out processes into sub-programmes can make reuse of code possible, if many methods carry out similar operations. Also, by specifying what a sub-program has to do, a developer can concentrate on using the sub program while others can worry about implementing it. Putting specific algorithms into sub-programmes can also increase readability and understandability of the code.

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

Created by Matyas Mehn — *Matyas Mehn [mailto:matyas.mehn@gmail.com] 2014/04/02 12:45*
