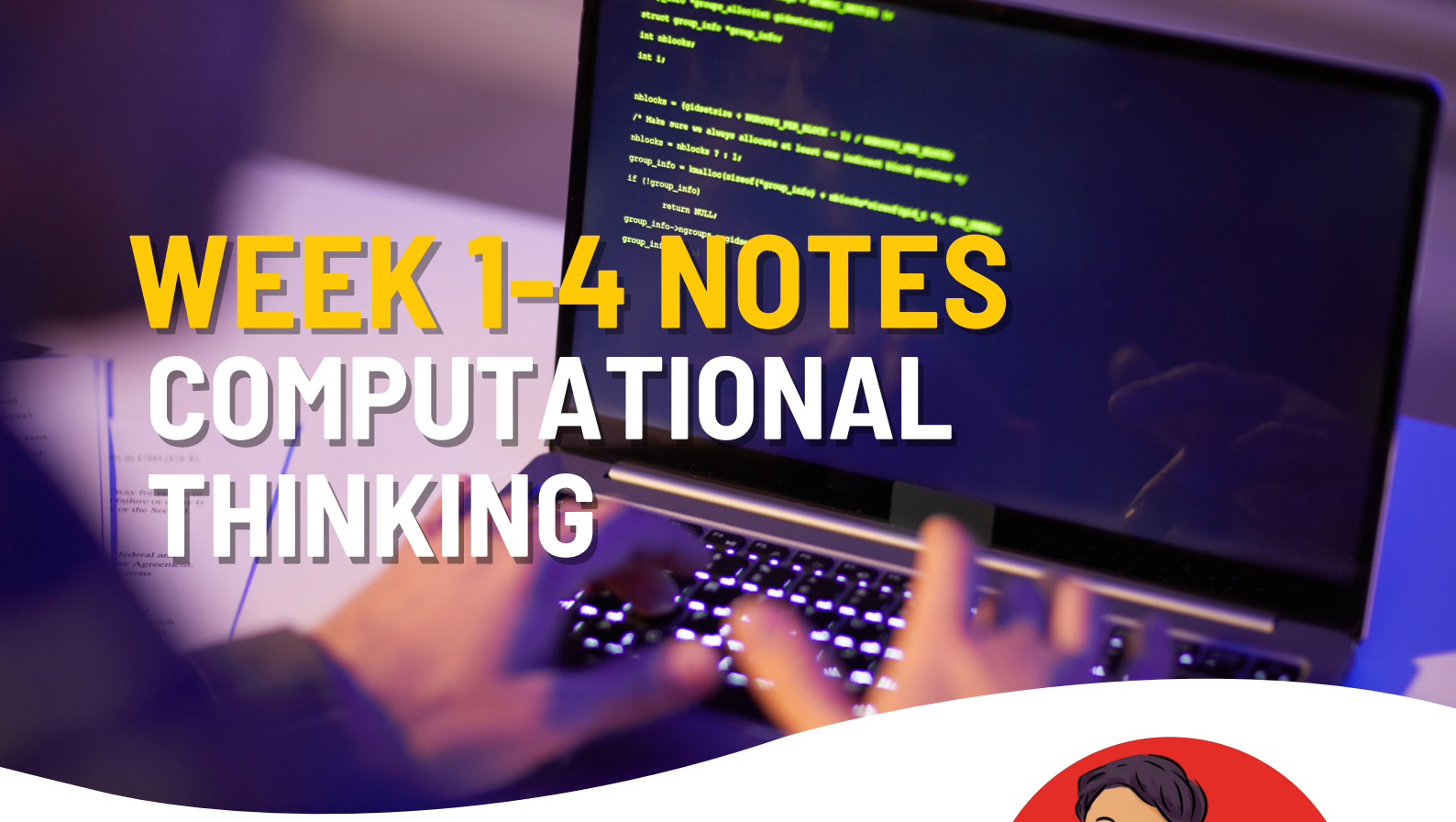


WEEK 1-4 NOTES
COMPUTATIONAL THINKING

INDEX

By :- Rushikesh Chavan

WEEK-1

By :- Rushikesh Chavan

Computational Thinking

Week 1

Flow Charts

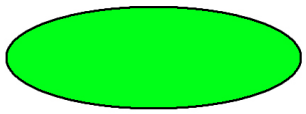
What is a Flowchart?

Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.

The process of drawing a flowchart for an algorithm is known as “flowcharting”.

Basic Symbols used in Flowchart Designs

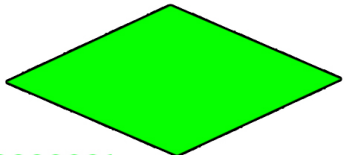
1. Terminal : The oval symbol indicates Start , Stop and Halt in a program ’s logic flow . A pause / halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



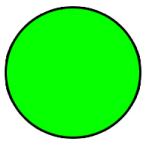
- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.

- **Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.

- **Decision** Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



- **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.



- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.

Advantages of Flowchart:

- Flowcharts are better way of communicating the logic of system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts helps in debugging process.
- With the help of flowcharts programs can be easily analyzed.
- It provides better documentation.
- Flowcharts serve as a good proper documentation.

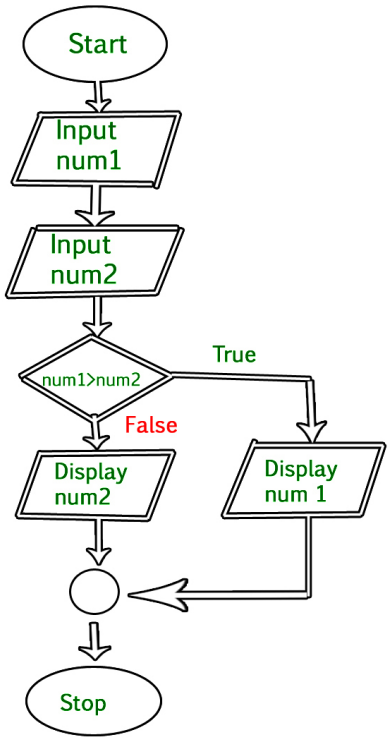
Disadvantages of Flowchart:

- It is difficult to draw flowchart for large and complex programs.
- In this their is no standard to determine the amount of detail.

- Difficult to reproduce the flowcharts.
- It is very difficult to modify the Flowchart.

Example : Draw a flowchart to input two numbers from user and display the largest of two numbers

Example:



Sanity of Data

The sanity of data: what we observed
• We organized our data set into cards, each storing one data item
• Each card had a number of elements, e.g.:
• numbers (e.g. marks)
• sequence of characters (e.g. name, bill item, word, etc)
• We observed that there were restrictions on the values each element can take:
• for example marks has to lie between 0 and 100
• name cannot have funny characters
• Constraints on the kinds of operations that can be performed:
• addition of marks is possible
• but a multiplication of marks does not make sense!

- compare one name with another to generate a boolean type (True or False)
- but cannot add a name with another!

Data Types

Data types are of 3 kinds

1. Charecter - Alpha-Numerics, Special Symbols - We can't perform any operations on this type of data - Result type - undefinded
2. Integers - Numerics range from Minus infinty to plus infinity - operations +,-,*,/,%, <,> - Result type: Integer or boolean
3. Boolean -True or False - operations AND, OR - result type Boolean

Subtypes:

▼ Integers:

Dates, Marks, Quantity, Ranks, count

▼ Charecter:

Gender

▼ Strings:

Names, City Words, Catagory

4. Record - Data type with multiple fields - each of which has a name and a value (Struct or Tuple)

▼ Examples of Record:

Marks card , Words in a Paragraph , Shopping bills

List

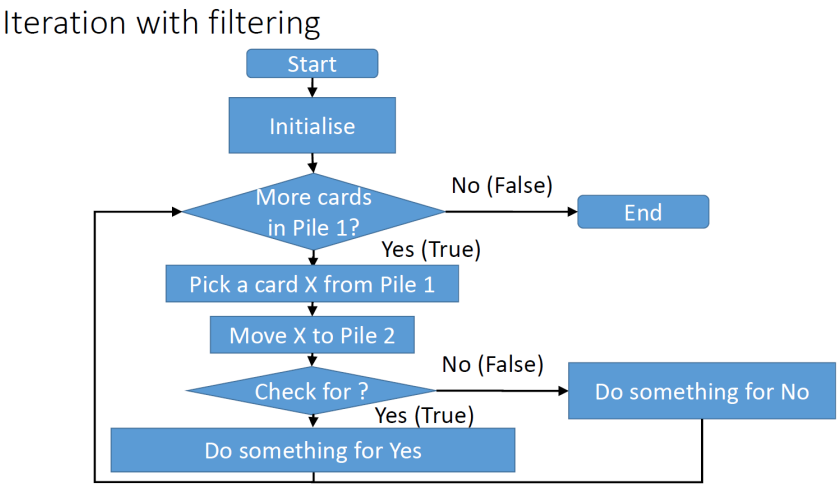
- A sequence of data elements (for example a sequence of records)
- MarksCardList - is the data type for our data set of all marks cards
- Each element in the sequence is of MarksCard Record data type
- ParagraphWordList - is the data type for our word data set
- Each element in the sequence is of WordInPara Record data type
- ShoppingBillList - data type for the shopping bill data set
- We need to define the Record data type for a shopping bill

WEEK-2

By :- Rushikesh Chavan

Computational Thinking

Week 2



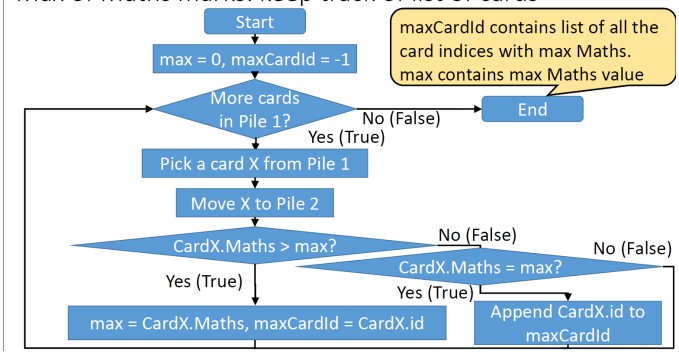
▼ To find Max marks

1. To Find max marks: Replace initialise with **Max=0**
2. Replace Check for with **Maths marks of Card X > max?** if so, **update max**

▼ To find Maths max marks

1. Initialise to **MaxCardId=-1**
2. Replace do something to **max = CardX.Maths, maxCardId = CardX.id**

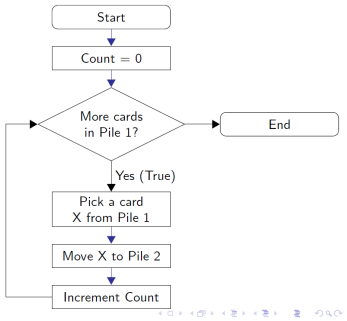
Max of Maths marks: keep track of list of cards



Pseudocode

Start
Count = 0
while (Pile 1 has more cards) {
  Pick a card **X** from Pile 1
  Move **X** to Pile 2
  Increment **Count**
}
End

1. Assign a value to a variable
2. Repeat steps while condition holds
3. Mark start and end of repeated block



Pseudocode: From pictures to text 6 / 7

Sum of Boys' Maths marks

```
Sum = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.Gender == M) {
    Sum = Sum + X.Maths
  }
}
```

- Conditional execution, once
- Equality (==) vs assignment (=)

Sum of Boys' and Girls' Maths mark

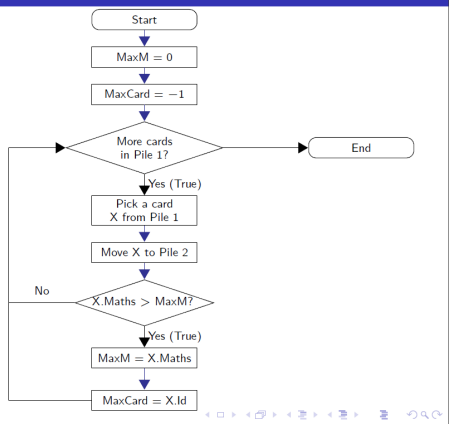
```
BoySum = 0
GirlSum = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.Gender == M) {
    BoySum = BoySum + X.Maths
  }
  else {
    GirlSum = GirlSum + X.Maths
  }
}
```

Finding the maximum Maths marks

```
MaxM = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.Maths > MaxM) {
    MaxM = X.Maths
  }
}
```

Finding the card with maximum Maths marks

```
MaxM = 0
MaxCard = −1
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.Maths > MaxM) {
    MaxM = X.Maths
    MaxCard = X.Id
  }
}
```



WEEK-3

By :- Rushikesh Chavan

Computational Thinking

Week 3

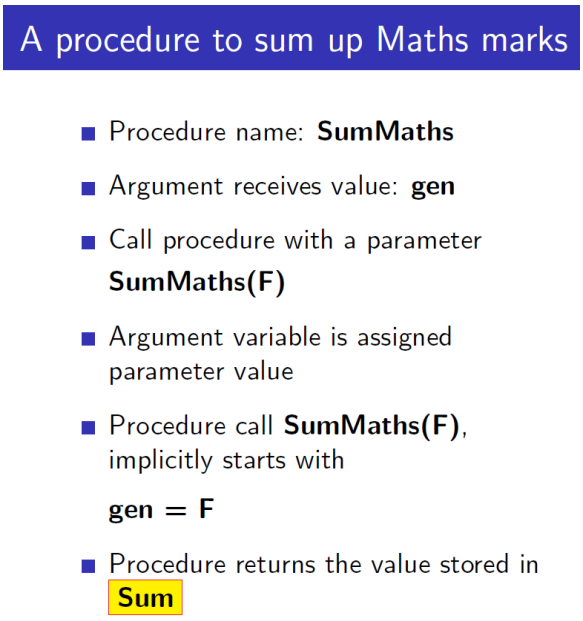
Extraction of data and Creation of tables

1.Data on cards can be naturally represented using tables
2.Each attribute is a column in the table
3.Each card is a row in the table
4.Difficulty if the cards has a variable number of attributes Items in shopping bill
5.Multiple rows | duplication of data
6.Split as separate tables and need to link via unique attribute
Tables

Procedures

A Procedure is a block of organized, reusable code that is used to perform a single, related action. Procedure provide better modularity for your application and a high degree of code reusing.

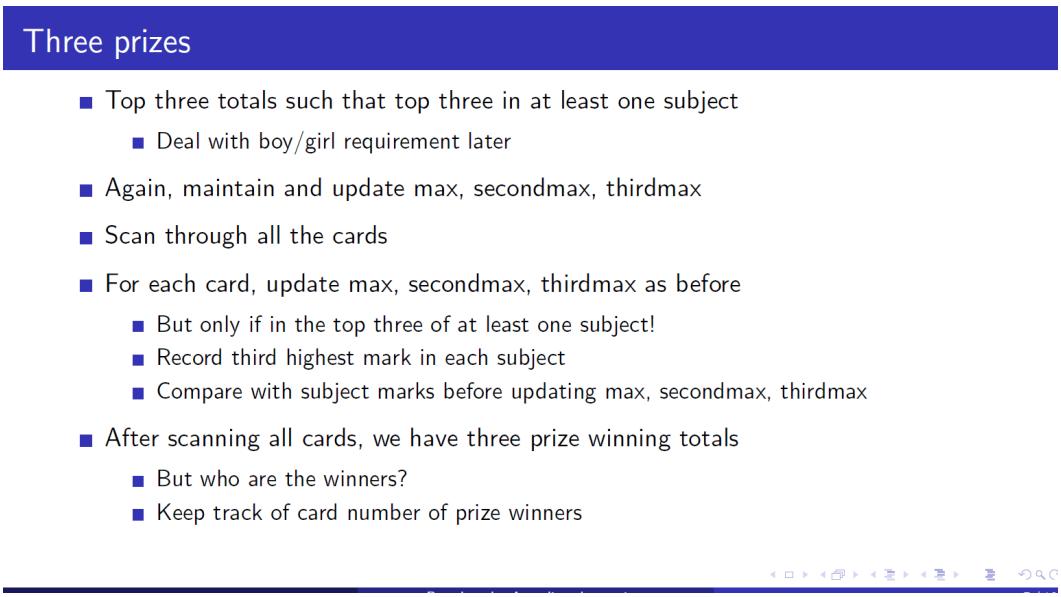
Example:



▼ To call a Procedure,

- use the Procedure name followed by parenthesis

1. A procedure may not return a value
2. Procedure call is a separate statement
3. Use a procedure when the same computation is used for differentsituations
4. Parameters fix the context
5. Use variables to save values returned by procedures
6. Keep track of the outcomes of multiple procedure calls
7. Procedures help to modularize pseudocode
8. Avoid descibing the same process repeatedly
9. If we improve the code in a procedure, benet automatically applies to all procedure calls



.

Three prizes

- Maintain max, secondmax, thirdmax, as well as maxid, secondmaxid, thirdmaxid
- Record third highest mark in each subject
- Scan through all the cards
- Update max, secondmax, thirdmax as appropriate
  - Only if top three in some subject — new procedure **SubjectTopper(...)**
- In the end, we have what we need

```
⟨ Initialization of max, maxid etc ⟩
⟨ Record third highest per subject ⟩
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  ⟨ Update max, maxid etc ⟩
  }
}
```

Variables of interest

- maxid, max
- secondmaxid, secondmax
- thirdmaxid, thirdmax

Three prizes, in entirety

```
max = 0
secondmax = 0
thirdmax = 0
maxid = -1
secondmaxid = -1
thirdmaxid = -1
maths3 = TopThreeMarks(Maths)
phys3 = TopThreeMarks(Physics)
chem3 = TopThreeMarks(Chemistry)
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  if (SubjectTopper(X,math3,phys3,chem3)){
    if (X.Total > max) {
      thirdmax = secondmax
      thirdmaxid = secondmaxid
      secondmax = max
      secondmaxid = maxid
      max = X.Total
      maxid = X.Id
    }
    if (max > X.Total > secondmax) {
      thirdmax = secondmax
      thirdmaxid = secondmaxid
      secondmax = X.Total
      secondmaxid = X.Id
    }
    if (secondmax > X.Total > thirdmax) {
      thirdmax = X.Total
      thirdmaxid = X.Id
    }
  }
}
```

Boundary conditions

- What if all prize winners are of the same gender?
- Exclude the third prize winner and repeat the process
  - How many times?
  - Till we get three prize winners with at least one boy and one girls
  - Will this always given us three valid prize winners?
- What if there are ties?
  - How many ties can we tolerate?
  - Does it depend on first, second or third position?

- We have worked out a complex problem in full detail
- Identify natural units to convert into procedures
  - TopThreeMarks(Subj)
  - SubjectTopper(CardId,MMark,PMark,CMark)
- Shortcut for checking return value of a procedure that returns a Boolean value
  - if (SubjectTopper(CardID,Math3,Phys3,Chem3))
- Have to anticipate and account for unexpected situations in data
  - All toppers are same gender
  - Ties

Side effects

- What is the status of **Deck** after the procedure?
- Is each card the same as it was before?
  - We certainly expect so
- Is the sequence of cards the same as it was before?
  - Perhaps not
  - Depends what we mean by "restore" **Deck**
  - **SeenDeck** would normally be in reverse order
- Side effect Procedure modifies some data during its computation

Interface vs implementation

Each procedure comes with a contract

- Functionality
  - What parameters will be passed
  - What is expected in return
- Data integrity
  - Can the procedure have side effects?
  - Is the nature of the side effect predictable?
    - For instance, deck is reversed

Contract specifies interface

- Can change procedure implementation (code) provided interface is unaffected

WEEK-4

By :- Rushikesh Chavan

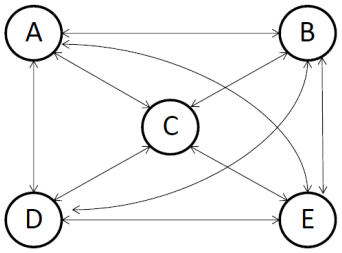
Computational Thinking

Week 4

Binning method is used to smoothing data or to handle noisy data. In this method, the data is first sorted and then the sorted values are distributed into a number of buckets or bins. As binning methods consult the neighborhood of values, they perform local smoothing.

Example

Comparing each element with all other elements



For 5 elements A, B, C, D, E:
The comparisons required are:

A with B, A with C, A with D, A with E	(4)
B with C, B with D, B with E	(3)
C with D, C with E	(2)
D with E	(1)

Number of comparisons: 4 + 3 + 2 + 1 = 10

- For N objects, the number of comparisons required will be:
  - (N - 1) + (N - 2) + ... + 1
  - which is = $\frac{N \times (N - 1)}{2}$
- This is the same as the number of ways of choosing 2 objects from N objects:
  - ${}^{N}C_{2} = \frac{N \times (N - 1)}{2}$
- From first principles:
  - Total number of pairs is N × N
  - From this reduce self comparisons (e.g. A with A). So number is reduced to: N × N - N
  - which can be written as N × (N - 1)
  - Comparing A with B is the same as comparing B with A, so we are double counting this comparison
  - So, reduce the count by half = $\frac{N \times (N - 1)}{2}$

Number of comparisons can be written as: ½ × N × (N - 1)

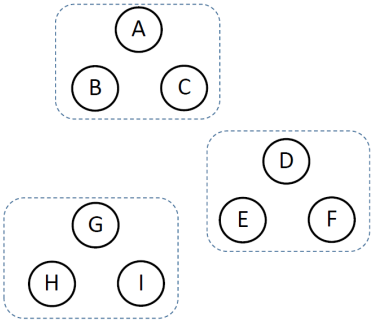
Calculation of reduction due to binning

- For N items:
- Number of comparisons without binning is: ½ × N × (N - 1)

- If we use K bins of equal size, number of items in each bin is: N/K
- Number of comparisons per bin is: ½ × N/K × (N/K - 1)
- Total number of comparisons is:
  K × ½ × N/K × (N/K - 1) = ½ × N × (N/K - 1)

- Factor of reduction is: [½ × N × (N - 1)] / [½ × N × (N/K - 1)]
  = (N - 1) / (N/K - 1)
- For N = 9 and K = 3, this is (9 - 1) / (3 - 1) = 4
  - So reduction is by a factor of 4 times.

Key idea: Use binning



- For 9 objects A,B,C,D,E,F,G,H,I:
  - The number of comparisons is ½ × 9 × (9 - 1)
    = ½ × 9 × 8 = 9 × 4 = 36

- If the objects can be binned into 3 bins of 3 each:
  - The number of comparisons per bin is:
    ½ × 3 × (3 -1) = ½ × 3 × 2 = 3
  - Total number of comparisons for all 3 bins is:
    3 × 3 = 9

- So, the number of comparisons reduces from 36 to 9 !
  - *Reduced by a factor of 4 times.*

Summary of concepts introduced in the course

What is computational thinking?

- Expressing problem solutions as a sequence of steps for communication to a computer
- Finding common patterns between solutions, apply these patterns to solve new problems

What is computational thinking?

- Expressing problem solutions as a sequence of steps for communication to a computer
- Finding common patterns between solutions, apply these patterns to solve new problems
- Data science problems are usually posed on a dataset
  - which can be obtained from real life artefacts - time tables, shopping bills, transaction logs
  - ... or may be available in a digital format - typically in the form of tables

What is computational thinking?

- Expressing problem solutions as a sequence of steps for communication to a computer
- Finding common patterns between solutions, apply these patterns to solve new problems
- Data science problems are usually posed on a dataset
  - which can be obtained from real life artefacts - time tables, shopping bills, transaction logs
  - ... or may be available in a digital format - typically in the form of tables
- Computational thinking in datascience involves finding patterns in methods used to process these datasets
- Through this course, several concepts and methods were introduced for doing this
  - Typically involves first scanning the dataset to collect relevant information
  - Then processing this information to find relationships between data elements
  - Finally organising the relationships in a form that allows questions to be answered easily

Iterators and Variables

- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
  - Iterator needs to be initialised
  - The steps that need to be repeated need to be made precise
  - We should know when and how to exit from an iteration, and where to go after the exit

Iterators and Variables

- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
  - Iterator needs to be initialised
  - The steps that need to be repeated need to be made precise
  - We should know when and how to exit from an iteration, and where to go after the exit
- **Variables** are storage units used to keep track of intermediate values during the iteration

Iterators and Variables

- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
  - Iterator needs to be initialised
  - The steps that need to be repeated need to be made precise
  - We should know when and how to exit from an iteration, and where to go after the exit
- **Variables** are storage units used to keep track of intermediate values during the iteration
- Initialisation and updates of variables are done through **assignment statements**

Iterators and Variables

- The most powerful construct to scan the dataset or to process intermediate information is the **iterator**.
- Whenever some task needs to be done repeatedly, an iterator is required
  - Iterator needs to be initialised
  - The steps that need to be repeated need to be made precise
  - We should know when and how to exit from an iteration, and where to go after the exit
- **Variables** are storage units used to keep track of intermediate values during the iteration
- Initialisation and updates of variables are done through **assignment statements**
- Variables which assemble a value or a collection are called **accumulators**

Pseudocode and flowchart for processing a dataset

Initialise variables

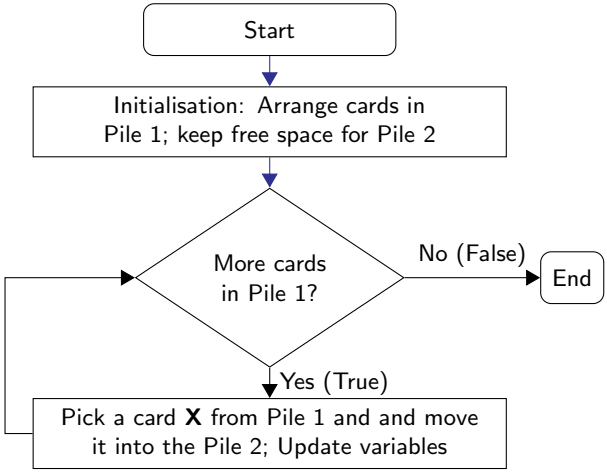
while (Pile 1 has more cards) {

  Pick a card **X** from Pile 1

  Move **X** to Pile 2

  Update values of variables

}



The set of items need to have well defined values

- Variables can be of different **datatypes**
- Basic data types: **boolean, integer, character** and **string**
- **Subtypes** put more constraints on the values and operations allowed
- Lists and Records are two ways of creating bigger bundles of data
- In a **list** all data items typically have the same datatype
- Whereas, a **record** has multiple named fields, each can be of a different datatype
- A **Dictionary** is like a record to which we can add new fields

Iteration with Filtering

- **Filtering** allows us to select the relevant data elements for processing and ignore the rest
- Requires complex boolean conditions to be defined
  - We can make compound conditions using boolean connectives - and, or, not
  - ... or we can do condition checking in sequence
  - ... or we can make nested condition checks

Iteration with Filtering

- **Filtering** allows us to select the relevant data elements for processing and ignore the rest
- Requires complex boolean conditions to be defined
  - We can make compound conditions using boolean connectives - and, or, not
  - ... or we can do condition checking in sequence
  - ... or we can make nested condition checks
- The conditions can work on the (fixed) data elements or on variables:
  - Compare the item values with a constant - example count, sum. The filtering condition does not change after each iteration step
  - compare item values with a variable - example max. The filtering condition changes after an iteration step

Iteration with Filtering

- **Filtering** allows us to select the relevant data elements for processing and ignore the rest
- Requires complex boolean conditions to be defined
  - We can make compound conditions using boolean connectives - and, or, not
  - ... or we can do condition checking in sequence
  - ... or we can make nested condition checks
- The conditions can work on the (fixed) data elements or on variables:
  - Compare the item values with a constant - example count, sum. The filtering condition does not change after each iteration step
  - compare item values with a variable - example max. The filtering condition changes after an iteration step
- Using filtering with accumulation, we can assemble a lot of intermediate information about the dataset

Procedures and parameters

- Sometimes we have to write the same piece of code again and again with small differences
- A piece of pseudocode can be converted into a **procedure** by separating it out from the rest of the code
- Some variables (or constants) used in this piece of code can be replaced by a **parameter**, so that the same procedure can be called with different parameter values to work on different data elements

Procedures and parameters

- Sometimes we have to write the same piece of code again and again with small differences
- A piece of pseudocode can be converted into a **procedure** by separating it out from the rest of the code
- Some variables (or constants) used in this piece of code can be replaced by a **parameter**, so that the same procedure can be called with different parameter values to work on different data elements
- A procedure can return a value, and the return value can be assigned to a variable. This makes the code much more compact and easy to read and manage

Procedures and parameters

- Sometimes we have to write the same piece of code again and again with small differences
- A piece of pseudocode can be converted into a **procedure** by separating it out from the rest of the code
- Some variables (or constants) used in this piece of code can be replaced by a **parameter**, so that the same procedure can be called with different parameter values to work on different data elements
- A procedure can return a value, and the return value can be assigned to a variable. This makes the code much more compact and easy to read and manage
- The procedure can have **side-effects**
  - it can change the value of variables that are passed as parameters
  - or those that are made accessible to the procedure, such as the data set elements or lists and dictionaries created from them
  - Procedures with side-effects need to be used carefully

Multiple iterations

- Two iterations can be carried out in sequence or nested

Multiple iterations

- Two iterations can be carried out in sequence or nested
- In a sequential iteration, we make multiple passes through the data, using the result of one pass during the next pass.
  - first pass could collect some intermediate information, and the second pass can filter elements using this information
  - It establishes a relation between elements with the aggregate
  - e.g. find all the below average students

Multiple iterations

- Two iterations can be carried out in sequence or nested
- In a sequential iteration, we make multiple passes through the data, using the result of one pass during the next pass.
  - first pass could collect some intermediate information, and the second pass can filter elements using this information
  - It establishes a relation between elements with the aggregate
  - e.g. find all the below average students
- Nested iterations are used when we want to create a relationship between pairs of data elements
  - Nested iterations are costly in terms of number of computations required
  - We could reduce the number of comparisons by using **binning** wherever possible
  - The relationships produced through nested iterations can be stored using lists, dictionaries (or **graphs**)

Lists

- A **list** is a sequence of values
- Write a list as `[x1,x2,...,xn]`, combine lists using `++`
  - `[x1,x2] ++ [y1,y2,y3]` $\mapsto$ `[x1,x2,y1,y2,y3]`
- Extending list `l` by an item `x`
  - `l = l ++ [x]`
- `foreach` iterates through values in a list
- `length(l)` returns number of elements in `l`
- Functions to extract first and last items of a list
  - `first(l)` and `rest(l)`
  - `last(l)` and `init(l)`

Sorted lists

- **Sorting** is an important pre-processing step
- **Insertion sort** is a natural sorting algorithm
  - Repeatedly insert each item of the original list into a new sorted list
  - The list can be sorted in ascending or descending order
- Sorted lists allow simpler solutions to be found to some problems - example identify the quartiles for awarding grades

Dictionaries

- A **dictionary** stores a collection of key:value pairs
- Random access — getting the value for any key takes constant time
- Dictionary is sequence `{k1:v1, k2:v2, ..., kn:vn}`
- Usually, create an empty dictionary and add key-value pairs

```
d = {}
d[k1] = v1
d[k7] = v7
```

- Iterate through a dictionary using `keys(d)`

```
foreach k in keys(d) {
  1 Do something with d[k]
}
```

- `isKey(d,k)` reports whether `k` is a key in `d`

```
if isKey(d,k){
  1d[k] = d[k] + v
}
else{
  1d[k] = v
}
```

Graphs

- **Graphs** are a useful way to represent relationships
  - Add an edge from `i` to `j` if `i` is related to `j`

Graphs

- **Graphs** are a useful way to represent relationships
  - Add an edge from `i` to `j` if `i` is related to `j`
- Use matrices to represent graphs
  - `M[i][j] = 1` — edge from `i` to `j`
  - `M[i][j] = 0` — no edge from `i` to `j`

Graphs

- **Graphs** are a useful way to represent relationships
  - Add an edge from `i` to `j` if `i` is related to `j`
- Use matrices to represent graphs
  - `M[i][j] = 1` — edge from `i` to `j`
  - `M[i][j] = 0` — no edge from `i` to `j`
- Iterate through matrix to aggregate information from the graph
- Graphs are useful to represent connectivity in a network
  - A path is a sequence of edges
  - Starting with direct edges, we can iteratively find longer and longer paths

Graphs

- **Graphs** are a useful way to represent relationships
  - Add an edge from `i` to `j` if `i` is related to `j`
- Use matrices to represent graphs
  - `M[i][j] = 1` — edge from `i` to `j`
  - `M[i][j] = 0` — no edge from `i` to `j`
- Iterate through matrix to aggregate information from the graph
- Graphs are useful to represent connectivity in a network
  - A path is a sequence of edges
  - Starting with direct edges, we can iteratively find longer and longer paths
- We can represent extra information in a graph via **edge labels** - e.g. distance
  - Iteratively update labels - e.g. compute shortest distance path between each pair of stations

Matrices

- **Matrices** are two dimensional tables
  - Support random access to any element `m[i][j]`

Matrices

- **Matrices** are two dimensional tables
  - Support random access to any element `m[i][j]`
- We can implement matrices using nested dictionaries

Matrices

- **Matrices** are two dimensional tables
  - Support random access to any element `m[i][j]`
- We can implement matrices using nested dictionaries
- Use iterators to process matrices row-wise and column-wise
  - `foreach r in rows(mymatrix)`
  - `foreach c in columns(mymatrix)`

Recursion

- Many functions are naturally defined in an inductive manner
  - Base case and inductive step

Recursion

- Many functions are naturally defined in an inductive manner
  - Base case and inductive step
- Use **recursive procedures** to compute such functions
  - Base case: value is explicitly calculated and returned. Has to be properly defined to ensure that the recursion terminates
  - Inductive case: value requires procedure to evaluated on a smaller input
  - Suspend the current computation till the recursive computation terminates

All the best for your exams, and for the rest of the programme !