Computational Thinking

Week 1

Flow Charts

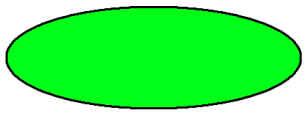
What is a Flowchart?

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

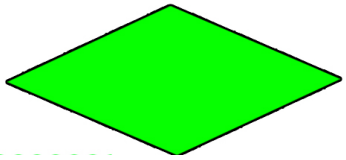
1. **Terminal:** The oval symbol indicates Start, Stop and Halt in a program’s logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



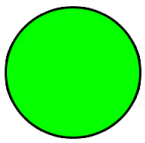
- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.

- **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.

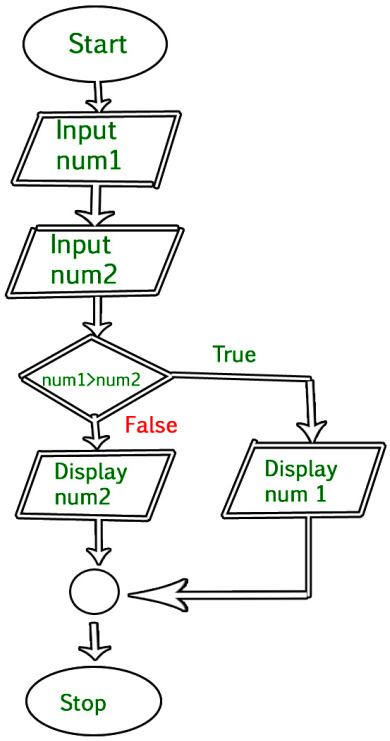
Disadvantages of Flowchart:

- 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:

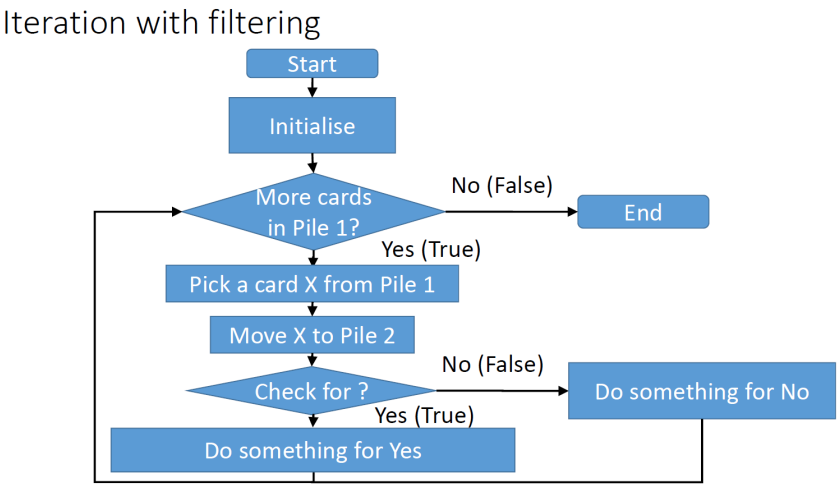
Marks card , Words in a Paragraph , Shopping bills

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

Computational Thinking

Week 2



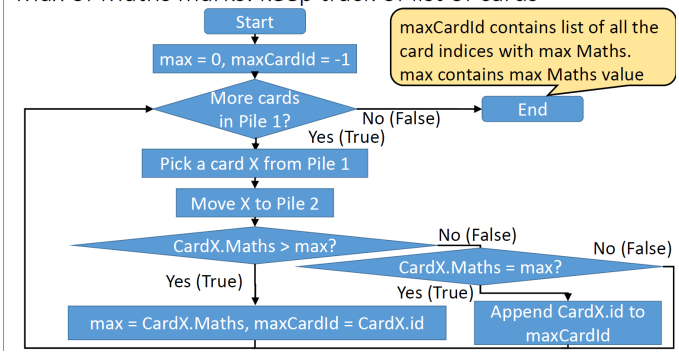
▼ To find Max marks

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**

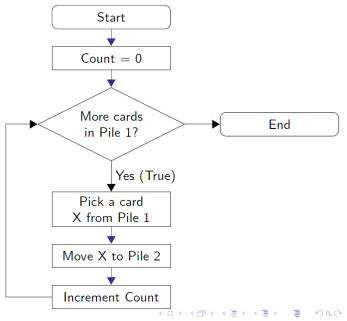
Max of Maths marks: keep track of list of cards



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

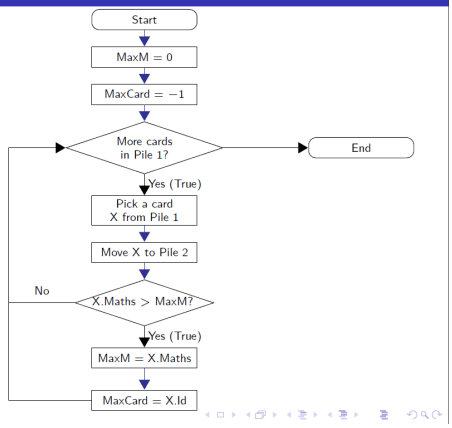
```
BoySum = 0
GirlSum = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.Gender == M) {
    BoySum = BoySum + X.Maths
  }
  else {
    GirlSum = GirlSum + X.Maths
  }
}
```

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
  }
}
```



Computational Thinking

Week 3

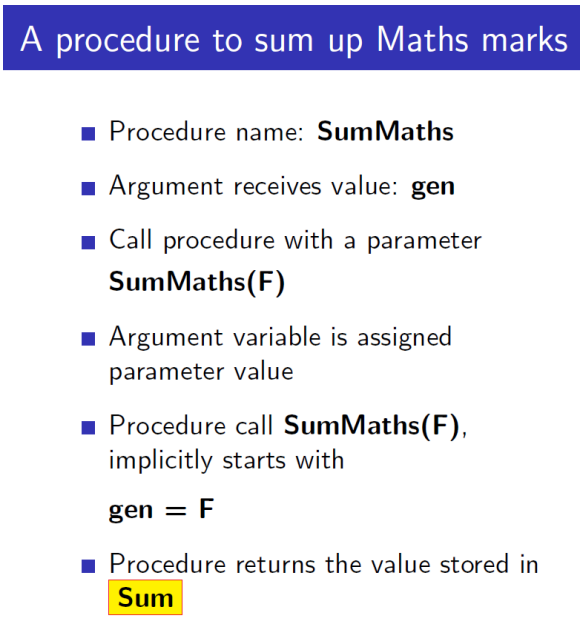
Extraction of data and Creation of tables

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.

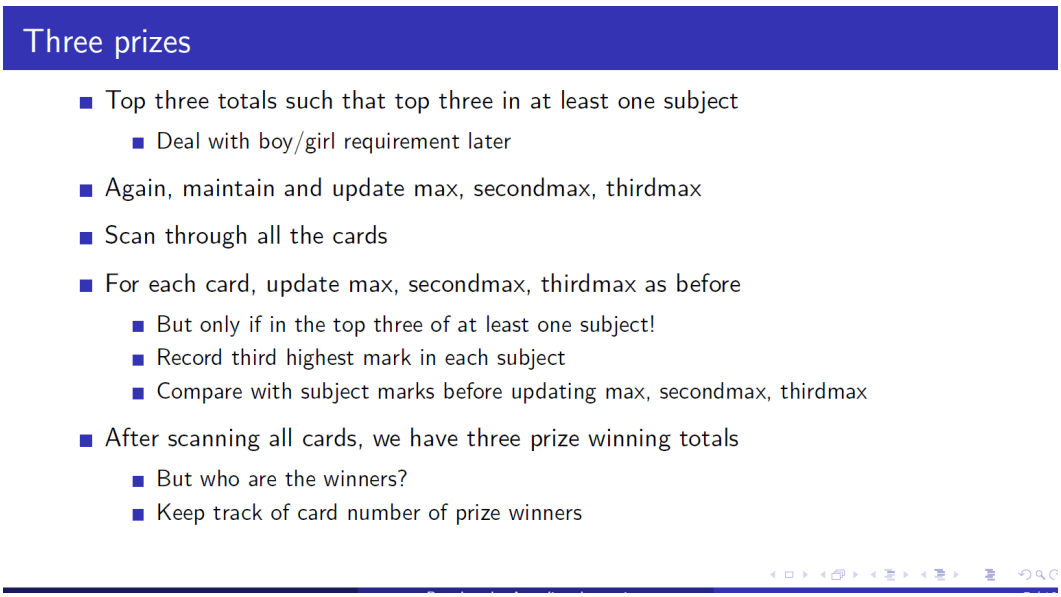
Example:



▼ 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

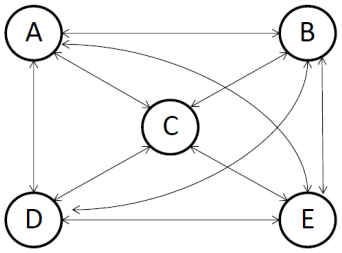
Computational Thinking

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)

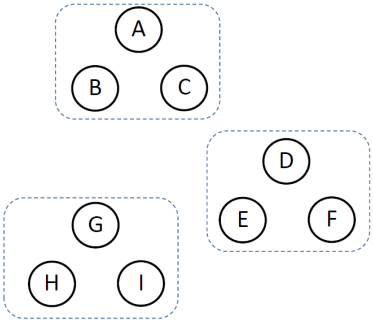
Calculation of reduction due to binning

- 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.*

Computational Thinking

Week 5

Collections

Collections are containers that are used to store collections of data, for example, list, dict, set, tuple etc. These are built-in collections. Several modules have been developed that provide additional data structures to store collections of data

- Variables keep track of intermediate values
- Often we need to keep track of a collection of values
- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered and changeable. No duplicate members.

▼ **Examples**

- Students with highest marks in Physics
- Customers who have bought food items from SV Stores
- Nouns that follow an adjective

▼ **Simplest collection is a list**

- Sequence of values
- Single variable refers to the entire sequence
- Notation for lists
- Primitive operations to manipulate lists

Pseudocode for lists

- Sequence within square brackets
  - `[1,13,2]`
  - `["Vedanayagam","cane", "Monday","school"]`
  - `[]` — empty list
- Append two lists, `l1 ++ l2`
  - `l1` is `[1,13]`, `l2` is `[2,17,1]`
  - `l1++l2` is `[1,13,2,17,1]`
- Extend `l` with item `x`
  - `l = l ++ [x]`
- Examples
  - List of students born in May
  - List of students from Chennai

```
chennaiList = []
while (Table 1 has more rows) {
   Read the first row X in Table 1
   if (X.TownCity == "Chennai") {
     chennaiList = chennaiList ++
                     [X.Seqno]
   }
   Move X to Table 2
}
```

Processing lists

- Typically, we need to iterate over a list
  - Examine each item
  - Process it appropriately
- `foreach x in l {`
  Do something with `x`
  `}`
  - `x` iterates through values in `l`
- Example
  - All students born in May who are from Chennai
  - Nested `foreach`

```
mayChennaiList = []
foreach x in mayList {
   foreach y in chennaiList {
     if (x == y) {
       mayChennaiList =
            mayChennaiList ++ [x]
     }
   }
}
```

Examples :

Identifying top students

- Find students who are doing well in all subjects
  - Among the top 3 marks in each subject
- Procedure for third highest mark in a subject
- Use lists
  - Construct a list of top students in each subject
  - Identify students who are present in all three lists

```
Procedure TopThreeMarks(Subj)
   max = 0, secondmax = 0, thirdmax = 0
   while (Table 1 has more rows) {
     Read the first row X in Table 1
     if (X.Subj > max) {
       thirdmax = secondmax
       secondmax = max
       max = X.Subj
     }
     if (max > X.Subj and X.Subj > secondmax) {
       thirdmax = secondmax
       secondmax = X.Subj
     }
     if (secondmax > X.Subj and X.Subj > thirdmax) {
       thirdmax = X.subj
     }
     Move X to Table 2
   }
   return(thirdmax)
End TopThreeMarks
```

Constructing the lists

- Obtain cutoffs in each subject
- Initialize lists for each subject
- Scan each row
- For each subject, check if the marks are within the top three
- If so, append to the list for that subject

```
cutoffMaths = TopThreeMarks(Mathematics)
cutoffPhys = TopThreeMarks(Physics)
cutoffChem = TopThreeMarks(Chemistry)

mathsList = []
physList = []
chemList = []

while (Table 1 has more rows) {
   Read the first row X in Table 1
   if (X.Mathematics >= cutoffMaths) {
     mathsList = mathsList ++ [X.SeqNo]
   }
   if (X.Physics >= cutoffPhys) {
     physList = physList ++ [X.SeqNo]
   }
   if (X.Chemistry >= cutoffChem) {
     chemList = chemList ++ [X.SeqNo]
   }
   Move X to Table 2
}
```

Find the overall toppers

- First find students who are toppers in Maths and Physics

```
mathsPhysList = []
foreach x in mathsList {
   foreach y in PhysList {
     if (x == y) {
       mathsPhysList = mathsPhysList ++ [x]
     }
   }
}
```

- Then match these toppers with toppers in Chemistry

```
mathsPhysChemList = []
foreach x in mathsPhysList {
   foreach y in chemList {
     if (x == y) {
       mathsPhysChemList =
                   mathsPhysChemList ++ [x]
     }
   }
}
```

▼ **Sorting a list often makes further computations simple**

- Finding the top k values
- Finding duplicates
- Grouping by percentiles | top quarter, next quarter, . . .

▼ **Insertion Sort**

- Create a second sorted list
- Start with an empty list
- Repeatedly insert next value from first list into correct position in the second list

Inserting into a sorted list

- We have a list `l` arranged in ascending order
- We want to insert a new element `x` so that the list remains sorted
- Move items from `l` to a new list till we find the place to insert `x`
- Insert `x` and copy the rest of `l`
- Be careful to handle boundary conditions
  - `l` is empty
  - `x` is smaller than everything in `l`
  - `x` is larger than everything in `l`

```
Procedure SortedListInsert(l,x)
  newList = []
  inserted = False

  foreach z in l {
    if (not(inserted)) {
      if (x < z) {
        newList = newList ++ [x]
        inserted = True
      }
    }
    newList = newList ++ [z]
  }

  if (not(inserted)) {
    newList = newList ++ [x]
  }

  return(newList)
End SortedListInsert
```

Insertion sort

- Once we know how to insert, sorting is easy
- Create an empty list
- Insert each element from the original list into this second list
- Return the second list
- Invariant — second list is always sorted
  - `[]` is sorted, since it is empty
  - Inserting into a sorted list returns a sorted list

```
Procedure InsertionSort(l)

  sortedList = []

  foreach z in l {
    sortedList =
       SortedListInsert(sortedlList,z)
  }

  return(sortedList)

End InsertionSort
```

Correlation of marks

- Assign grades A,B,C,D in both subjects
- Perform well in Maths" grade B or above

- Perform at least as well in Physics - Physics grade > Maths grade

▼ **Algorithm**

- Assign grades in each subject
- Construct lists of students with grades A and B in both subjects - four lists
- Count students in A list for Maths who are also in A list for Physics
- Count students in B list for Maths who are also in A list or B list for Physics
- Use these counts to conrm or reject the hypotheisis

Computational Thinking

Week 6

DICTIONARY

Dictionaries are used to store data values in key : value pairs. A dictionary is a collection that is ordered*, changeable, and does not allow duplicates.

▼ **INDEXED COLLECTIONS AND DICTIONARIES**

- A list keeps a sequence of values
- Can iterate through a list, but random access is not possible
- To get the value at position i , need to start at the beginning and walk down i-1 steps

▼ A Dictionary stores Key : Value pairs **Examples**

- Chemistry marks (value) for each student (key)
- Source station (value) of a train route (key)
- m = chemMarks["Rahul"]
- s = sourceStation["10215"]

▼ Syntax of passing values to dictionary

- syntax-{"key : value"} example- {"Rahul" : 92, "Ritika" : 89}
- Empty Dictionary - {}
- Access value by providing key within square brackets. Example - s = sourceStation["10215"]

▼ Assigning a value | replace value or
create new key-value pair

- chemMarks["Rahul"] = 92
- Dictionary must exist to create new entry. Example - Initialize as d = {}

Example

Collect Chemistry marks in a dictionary

```
chemMarks = {}
while (Table 1 has more rows) {
   Read the first row X in Table 1
     name = X.Name
     marks = X.ChemistryMarks
     chemMarks[name] = marks
   }
   Move X to Table 2
}
```

Processing dictionaries

- How do we iterate through a dictionary?
- `keys(d)` is the list of keys of `d`

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

- Example
  - Compute average marks in Chemistry

```
total = 0
count = 0
foreach k in keys(chemMarks) {
   total = total + chemMarks[k]
   count = count + 1
}
chemavg = total/count
```

▼ Checking for a key

- isKey (d,k) - returns **TRUE** if **K** is a key in **D** **FALSE** otherwise

▼ TYPICAL USAGE

```
if isKey(runs,"Kohli"){
  runs["Kohli"] = runs["Kohli"]
                  + score
}
else{
  runs["Kohli"] = score
}
```

- Implementing `isKeys(d,k)`
  - Iterate through `keys(d)` searching for the key `k`
- Takes time proportional to size of the dictionary
- Instead, assume `isKeys(d,k)` is given to us, works in constant time
  - Random access

```
Procedure isKey(D,k)
    found = False
    foreach key in keys(D) {
      if (key == k) {
        found = True
        exitloop
      }
    }
    return(found)
End isKey
```

Customers buying food items

- Find the customer who buying the highest amount of food items
- Create a dictionary to store food purchases
  - Customer names as keys
  - Number of food items purchased as values

```
foodD = {}
while (Table 1 has more rows) {
   Read the first row X in Table 1
   customer = X.CustomerName
   items = X.Items
   foreach row in items {
     if (row.Category == "Food") {
       if (isKey(foodD,customer)) {
         foodD[customer]
                = foodD[customer] + 1
       }
       else {
         foodD[customer] = 1
       }
     }
   }
   Move X to Table 2
}
```

Birthday paradox

- Find a birthday shared by more than one student
- Create a dictionary with dates of births as keys
- Record duplicates in a separate dictionary
- If we want to record the names of those who share the birthday, store a list of student ids against each date of birth
- Can also store the students associated with each date of birth as a dictionary

```
birthdays = {}
duplicates = {}
while (Table 1 has more rows) {
   Read the first row X in Table 1
   dob = X.DoB
   seqno = X.SeqNo
   if (isKey(birthdays,dob)) {
     duplicates[dob] = True
     birthdays[dob][seqno] = True
   }
   else {
     birthdays[dob] = {}
     birthdays[dob][seqno] = True
   }
   Move X to Table 2
}
```

Resolving pronouns

- Resolve each pronoun to matching noun
  - Nearest noun preceding the pronoun
- Create a dictionary with part of speech as keys, sorted list of card numbers as values.

```
partOfSpeech = {}
partOfSpeech['Noun'] = []
partOfSpeech['Pronoun'] = []

while (Table 1 has more rows) {
   Read the first row X in Table 1
   if (X.PartOfSpeech == 'Noun') [
     partOfSpeech['Noun'] =
         partOfSpeech['Noun']
         ++ [X.SerialNo]
   }
   if (X.PartOfSpeech == 'Pronoun') [
     partOfSpeech['Pronoun'] =
         partOfSpeech['Pronoun']
         ++ [X.SerialNo]
   }
   Move X to Table 2
}
```

Resolving pronouns

- Resolve each pronoun to matching noun
  - Nearest noun preceding the pronoun
- Create a dictionary with part of speech as keys, sorted list of card numbers as values.
- Iterate through the dictionary to match pronouns
- Note that `partOfSpeech['Noun']` and `partOfSpeech['Pronoun']` are both sorted in ascending order of `SerialNo`

```
matchD = {}
foreach p in partOfSpeech['Pronoun'] {
   matched = -1
   foreach n in partOfSpeech['Noun'] {
     if (n < p) {
       matched = n
     }
     else {
       exitloop
     }
   }
   matchD[p] = matched
}
```

Dealing with side-effects

- Comparing two lists for duplicate items
  - Nested loop
- What if the lists are sorted?
  - Need not start inner iteration from the beginning
  - Use `first()` and `rest()` to cut down the list to be scanned
- Second list has been modified inside the procedure
  - Side-effect!
- Instead, make a copy of the input parameter

```
Procedure FindOverlap3(l1,l2)
  overlap = []
  myl2 = l2
  foreach x in l1 {
    y = first(myl2)
    myl2 = rest(myl2)
    while (y < x){
      y = first(myl2)
      myl2 = rest(myl2)
    }
    if (x == y) {
      overlap = overlap ++ [x]
    }
  }
  return(overlap)
End FindOverlap3
```

Deleting a key from a dictionary

- Delete a key from a dictionary?
  - Copy all keys and values except the one to be deleted to a new dictionary
  - Copy back the updated dictionary
- In this case, the side effect in the procedure is intended
  - Use side-effects to update a collection inside a procedure
  - Sorting a list in place
- We can also program this without side-effects
  - Return the updated dictionary
  - Reassign it after the procedure call

```
Procedure DeleteKey2(d,k)
  myd = {}
  foreach key in keys(d) {
    if (k ≠ key) {
      myd[key] = d[key]
    }
  }
  return(myd)
End DeleteKey2

myd = DeleteKey2(myd,k)
```

Computational Thinking

Week 7

▼ Often we need a **MATRIX**

- a two or three dimensional table
- with **m** rows and **n** columns
- Random Access - **matrix [i] [j]**

▼ **Examples of Matrix implimentation**

-

Implementing matrices

- Dictionaries support random access
- Create a nested dictionary
  - Outer key corresponds to rows
  - Inner key corresponds to columns
- Create a matrix

```
mymatrix = CreateMatrix(30,45)
```

```
Procedure CreateMatrix(rows,cols)
  mat = {}
  i = 0
  while (i < rows) {
    mat[i] = {}
    j = 0
    while (j < cols){
      mat[i][j] = 0
      j = j + 1
    }
    i = i + 1
  }
  return(mat)
End CreateMatrix
```

-

Processing matrices

- Typically we need to process all elements, either row by row or column by column

```
for each row i of mymatrix {
  for each column j of mymatrix {
    Do something with mymatrix[i][j]
  }
}
```

- Iterating through the rows
  - Row indices are keys of outer dictionary
  - Column indices are keys of first (any) row
  - `keys(d)` produces a list in arbitrary order
  - Assume a suitable `sort()` procedure
  - `sort(keys(d))` — ascending order

```
foreach r in sort(keys(mymatrix)) {
  foreach c in sort(keys(mymatrix[0])) {
    Do something with mymatrix[r][c]
  }
}
```

To improve readability, use `rows()` and `columns()`

```
foreach c in columns(mymatrix) {
  foreach r in rows(mymatrix) {
    Do something with mymatrix[r][c]
  }
}
```

Can also process a matrix columnwise

Mentoring

- Student A can mentor student B in a subject if A has higher marks, but not too much higher are between
  - Difference is between 10 and 20 marks
- Create a dictionary for marks in subject
  - `mathMarks = ReadMarks(Mathematics)`
- Create a mentoring graph for a subject
  - Represent as a matrix
  - `M[i][j] = 1` — edge from `i` to `j`
  - `M[i][j] = 0` — no edge from `i` to `j`

```
Procedure CreateMentorGraph(marks)
  n = length(keys(marks))
  mentorGraph = CreateMatrix(n,n)
  foreach i in keys(marks){
    foreach j in keys(marks){
      ijMarksDiff = marks[i] - marks[j]
      if (10 ≤ ijMarksDiff and
            ijMarksDiff ≤ 20) {
        mentorGraph[i][j] = 1
      }
    }
  }
  return(mentorGraph)
End CreateMentorGraph(marks)
```

Pairing students in study groups

- A can mentor student B in one subject and B can mentor A in the other
- Study groups in Maths and Physics
  - Create mentoring graphs for each
- Use the mentoring graphs to pair off students

```
mathMarks = ReadMarks(Mathematics)
phyMarks = ReadMarks(Physics)

mathMentorGraph =
          CreateMentorGraph(mathMarks)
phyMentorGraph =
          CreateMentorGraph(phyMarks)

paired = {}

foreach i in rows(mathMentorGraph) {
  foreach j in columns(mathMentorGraph) {
    if (mathMentorGraph[i][j] == 1 and
          phyMentorGraph[j][i] == 1 and
          not(isKey(paired,i)) and
          not(isKey(paired,j))) {
      paired[i] = j
      paired[j] = i
    }
  }
}
```

Popular students

- A student who can be mentored by many other students is popular
- Create mentoring graphs for all three subjects
- Count incoming mentoring edges for each student
- Avoid duplicates
  - Explicitly keep track of mentors for each student and count them

```
mentors = {}
popularity = {}
foreach j in columns(mathMentorGraph) {
  mentors[j] = {}
  foreach i in rows(mathMentorGraph) {
    if (mathMentorGraph[i][j] == 1){
      mentors[j][i] = True
    }
    if (phyMentorGraph[i][j] == 1){
      mentors[j][i] = True
    }
    if (chemMentorGraph[i][j] == 1){
      mentors[j][i] = True
    }
  }
  popularity[j]
       = length(keys(mentors[j]))
}
```

Similar students

- Two students are similar if they have similar marks in all subjects
  - Difference is within 10 marks
- Dictionaries with marks in each subject

```
mathMarks = ReadMarks(Mathematics)
phyMarks = ReadMarks(Physics)
chemMarks = ReadMarks(Chemistry)
```

- Create a similarity graph

```
Procedure
    CreateSimilarityGraph(marks1,mark2,marks3)
  n = length(keys(marks1))
  similarityGraph = CreateMatrix(n,n)
  foreach i in keys(marks1){
    foreach j in keys(marks1){
      ijDiff1 = abs(marks1[i] - marks1[j])
      ijDiff2 = abs(marks2[i] - marks2[j])
      ijDiff3 = abs(marks3[i] - marks3[j])
      if (ijDiff1 ≤ 10 and
            ijDiff2 ≤ 10 and
            ijDiff3 ≤ 10)}
        similarityGraph[i][j] = 1
      }
    }
  }
  return(similarityGraph)
End CreateSimilarityGraph(marks)
```

Computational Thinking

WEEK 8

▼ Graph

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges. The various terms and functionalities associated with a graph is described in great detail in our tutorial here. In this chapter we are going to see how to create a graph and add various data elements to it using a python program. Following are the basic operations we perform on graphs.

- Display graph vertices
- Display graph edges
- Add a vertex
- Add an edge
- Creating a graph

A graph can be easily presented using the python dictionary data types. We represent the vertices as the keys of the dictionary and the connection between the vertices also called edges as the values in the dictionary.

▼ Trains dataset introduced in Week 8

- Each trains is a route list of stations
- Each station is a route list of stations passing through

▼ Representation

- trains [t] [start], trains [t] [end]

▼ Direct Routes

- Count only start and end stations and ignore intermediate stations of the train
- Example - given below

```
Procedure DirectRoutes(trains)
  stations = {}
  foreach t in keys(trains) {
    stations[trains[t][start]] = True
    stations[trains[t][end]] = True
  }
  n = length(keys(stations))
  direct = CreateMatrix(n,n)
  stnindex = {}
  i = 0
  foreach s in keys(stations) {
    stnindex[s] = i
    i = i+1
  }
  foreach t in keys(trains){
    i = stnindex[trains[t][start]]
    j = stnindex[trains[t][end]]
    direct[i][j] = 1
  }
  return(direct)
End DirectRoutes
```

▼ One hop routes

- Stations A and B such that you can reach B from A by changing one train
- Example - given below

```
Procedure WithinOneHop(direct)
  n = length(keys(direct))
  onehop = CreateMatrix(n,n)
  foreach i in rows(direct) {
    foreach j in columns(direct) {
      onehop[i][j] = direct[i][j]
      foreach k in columns(direct) {
        if (direct[i][k] == 1 and
            direct[k][j] == 1) {
          onehop[i][j] = 1
        }
      }
    }
  }
  return(onehop)
End WithinOneHop
```

▼ Two hop routes

- Stations A and B such that you can reach B from A by changing at most two trains
- Example - given below

```
Procedure WithinTwoHops(direct,onehop)
  n = length(keys(direct))
  twohops = CreateMatrix(n,n)
  foreach i in rows(direct) {
    foreach j in columns(direct) {
      twohops[i][j] = onehop[i][j]
      foreach k in columns(direct) {
        if (onehop[i][k] == 1 and
              direct[k][j] == 1) {
          twohops[i][j] = 1
        }
      }
    }
  }
  return(twohops)
End WithinTwoHops
```

▼ n hop routes

- Let n hops record connections from station A to station B by changing at most n trains
- Example - given below

```
Procedure OneMoreHop(direct,nhops)
  n = length(keys(direct))
  onemorehop = CreateMatrix(n,n)
  foreach i in rows(direct) {
    foreach j in columns(direct) {
      onemorehop[i][j] = nhops[i][j]
      foreach k in columns(direct) {
        if (nhops[i][k] == 1 and
              direct[k][j] == 1) {
          onemorehop[i][j] = 1
        }
      }
    }
  }
  return(onemorehop)
End OneMoreHop
```

▼ **Edge Labled graph**

▼ Create a matrix to record direct routes

- Compile the list of stations from trains
- Map stations to row, column indices
- Populate the matrix

▼ Keep track of trains connecting stations

- Each entry in the matrix is now a dictionary
- Initially empty dictionary - no direct connection
- Add a key for each train connecting a pair of stations

▼ Example 1

```
Procedure LabelledDirectRoutes(trains)
   :
   foreach r rows(direct) {
     foreach c columns(direct) {
       direct[i][j] = {}
     }
   }
   foreach t in keys(trains){
     i = stnindex[trains[t][start]]
     j = stnindex[trains[t][end]]
     direct[i][j][t] = True
   }
   return(direct)
End DirectRoutes2
```

Distances between stations

- For each direct train record the distance it travels
  - Add an extra key, `trains[t][distance]`
- Compute the shortest distance by direct trains
  - Trains may take different routes between the same pair of stations
- Graph `directdist`
  - Edge label `directdist[i][j]` is the shortest direct distance between `i` and `j`

Distances between stations

- Compute the shortest distance by direct trains
- Edge-labelled graph `directdist`
- When we discover a direct route for the first time, set the distance
- If we find a new direct route between an already connected pair, update the distance to the minimum

```
Procedure DirectDistance(trains)
  ⋮
  directdist = CreateMatrix(n,n)
  ⋮
  foreach t in keys(trains){
    i = stn2idx[trains[t][start]]
    j = stn2idx[trains[t][end]]
    if (directdist[i][j] == 0) {
      directdist[i][j] = trains[t][distance]
    }
    else
      directdist[i][j] =
          min(directdist[i][j],trains[t][distance])
    }
  }
  return(directdist)
End DirectDistance
```

One hop distance

- We start with the matrix of direct distances
- Initialize one hop distance to direct distance
- Each time we discover a one hop route, update the one hop distance
- Iterate this to find length of the shortest path between each pair of stations
  - Modify the transitive closure calculation to record minimum distance path

```
Procedure OneHopDistance(directdist)
  n = length(keys(directdist))
  onehopdist = CreateMatrix(n,n)
  foreach i in rows(directdist) {
    foreach j in columns(directdist) {
      onehopdist[i][j] = directdist[i][j]
      foreach k in columns(directdist) {
        if (directdist[i][k] > 0 and
            directdist[k][j] > 0) {
          newdist = directdist[i][k]
                    + directdist[k][j]
          if (onehopdist[i][j] > 0){
            onehopdist[i][j] =
              min(newdist,onehopdist[i][j])
          }
          else{
            onehopdist[i][j] = newdist
  } } } } }
  return(onehopdist)
End OneHopDistance
```

Computational Thinking

Week 9

Function **Recursion,** which means a defined function can **call itself**.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

▼ Many computations are naturally defined inductively

- Base case: directly return the value
- Inductive step: compute value in terms of smaller arguments

- Factorial
  - $n! = n \times (n-1) \times \cdots \times 2 \times 1$
  - 0! is defined to be 1
  - `factorial(0) = 1`
  - For `n > 0`, `factorial(n) = n * factorial(n-1)`

```
Procedure Factorial(n)
  if (n == 0) {
   return(1)
  }
  else {
   return(n * factorial(n-1))
  }
End Factorial
```

- Recursive procedure
  - `factorial(n)` is suspended till `factorial(n-1)` returns a value

Inductive definitions on lists

- Inductive functions on lists
  - Base case: Empty list
  - Inductive step: Compute value in terms first element and rest
- Sum of numbers in a list
  - If `l == []`, `sum` is `0`
  - Otherwise, add `first(l)` to sum of `rest(l)`
  - Can also add `last(l)` to sum of `init(l)`

```
Procedure Listsum2(l)
  if (l == []) {
   return(0)
  }
  else {
   return(last(l) +
          Listsum2(init(l)))
  }
End Listsum2
```

Insertion sort

- Build up a sorted prefix
- Extend the sorted prefix by inserting the next element in the correct position

Procedure InsertionSort(l)

```
sortedList = []
foreach z in l {
  sortedList =
     SortedListInsert(sortedlList,z)
}
return(sortedList)
```

End InsertionSort

Procedure SortedListInsert(l,x)

```
newList = []
inserted = False

foreach z in l {
  if (not(inserted)) {
    if (x < z) {
      newList = newList ++ [x]
      inserted = True
    }
  }
  newList = newList ++ [z]
}

if (not(inserted)) {
  newList = newList ++ [x]
}

return(newList)
```

End SortedListInsert

Insertion sort, inductively

- List of length 1 or less is sorted
- For longer lists, insert `first(l)` into sorted `rest(l)`

Procedure InsertionSort(l)

```
if (length(l) <= 1) {
  return(l)
}
else {
  return(
      SortedListInsert(
        InsertionSort(rest(l)),
        first(l)
      )
  )
}
```

End InsertionSort

Procedure SortedListInsert(l,x)

```
newList = []
inserted = False

foreach z in l {
  if (not(inserted)) {
    if (x < z) {
      newList = newList ++ [x]
      inserted = True
    }
  }
  newList = newList ++ [z]
}

if (not(inserted)) {
  newList = newList ++ [x]
}

return(newList)
```

End SortedListInsert

Reachability in graphs

- What are the vertices reachable from node `i`?
- Start from `i`, visit a neighbour `j`
- Suspend the exploration of `i` and explore `j` instead
- Continue till you reach a vertex with no unexplored neighbours
- Backtrack to nearest suspended vertex that still has an unexplored neighbour
- Best defined recursively
  - Maintain information about visited nodes in a dictionary `visited`
  - Recursively update `visited` each time we explore an unvisited neighbour

Depth First Search (DFS)

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

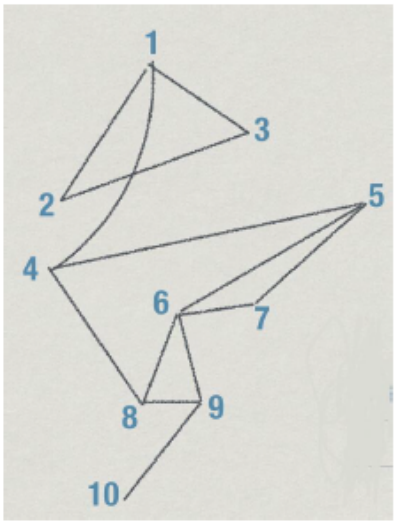
▼ **Depth First Search**

- Maintain information about visited nodes in a dictionary "visited"
- Recursively update visited each time we explore an unvisited neighbour
- To explore vertices reachable from i

- Initialize `visited = {}`
- `visited = DFS(graph,visited,i)`
- `keys(visited)` is set of nodes that can be reached from `i`
  - If `keys(visited)` includes all nodes, the graph is connected

```
Procedure DFS(graph,visited,i)
  visited[i] = True
  foreach j in columns(graph){
   if (graph[i][j] == 1 and
        not(isKey(visited,j))) {
     visited =
        DFS(graph,visited,j)
   }
  }
  return(visited)
End DFS
```

Example



- visited= { 4:True, 1:True, 2:True, 3:True, 5:True, 6:True, 7:True, 8:True, 9:True, 10:True }
- DFS(graph,visited,4)
- DFS(graph,visited,1)
- DFS(graph,visited,2)
- DFS(graph,visited,3)
- DFS(graph,visited,5)
- DFS(graph,visited,6)
- DFS(graph,visited,7)
- DFS(graph,visited,8)
- DFS(graph,visited,9)
- DFS(graph,visited,10)

Computational Thinking

Week 10

▼ **Encapsulation**

- Procedures that we have seen so far have been unanchored.
- It seems more natural to attach the procedure to the data elements on which it operates
- Encapsulate the data elements and the procedures into one package
- which is called an object, hence the popular term object oriented computing/programming
- The procedures encapsulated in the object act as the interfaces through which the external world can interact with the object
- The object can hide details of the implementation from the external world
- The changed implementation may involve additional (intermediate) data elements and additional procedures
- Any changes made to the implementation does not impact the external world, since the procedure interface is not changed
- Allows for separation of concerns - separate the "what?" from the "how?"

▼ **Procedures already provide some kind of encapsulation:**

- interface via parameters and return value
- hiding of variables used within the procedure

▼ **But we could have the following issues with procedures:**

- Procedures could have side-effects - some of them are desirable (recall the delete key example?)
- We may need to call a sequence of procedures to achieve something - we expect the object to hold the state between the procedure calls (ATM example)

▼ Recall of datatypes

- Basic datatypes - integer, character, boolean
- Subtype of a datatype to restrict the values and operations
- Records, Lists, Strings: compound datatypes
- Each card (table row) can be represented using a datatype
- The collection of cards can also be represented as a datatype

▼ Operations on a datatypes

- Well defined operations on the basic datatypes and their subtypes
- Some operations on the string datatype

But what about operations on compound datatypes?

What if we are allowed to attach our own procedures to a datatype?

- Allow a datatype to have procedure fields
- Just as X.F represents the field F of X, we can use X.F(a,b) to represent a procedure field of X which takes parameters a and b.

Example: Classroom Scores dataset

- Questions most frequently asked of the Scores dataset
  - What is the average in a subject - say Maths?
  - What is the overall average?
- Suppose that we have to ask these questions many times
  - It is wasteful to carry out the same computation again and again
  - Can we not store the answer of the question, and just return the saved answer when the question is asked again ?
- This will work as long as the dataset is static.

Example: encapsulation

- We could create an object CT (for class teacher) of datatype ClassAve
  - ClassAve needs only the list of total marks of the students
- ClassAve can have a field marksList that holds the total marks of all the students in the classroom dataset.
- We can now add a procedure average() to ClassAve to find the average of the list
- Since we want to store the answer after the first time, we could have another field aValue that will hold the computed value of average. Initially aValue = -1.
  - CT.average() first checks if CT.aValue is -1.
  - If no, it just returns CT.aValue
  - If it is -1, this means that the average has not been computed yet. So, it computes the average by summing the marks in the list and dividing the sum by the length of the list

Example: encapsulation

- What about the average values of the individual subjects?
- Just like CT, we could have objects PhT, MaT and ChT (for Physics, Maths and Chemistry Teachers) that each hold (or have access to) the entire classroom dataset.
  - But again as we observed with CT, PhT needs to hold only the list of Physics marks of the students, similarly MaT and ChT need only hold the Maths and Chemistry marks lists.
- So, let us say that each of these objects are also of the same datatype ClassAve, but their marksList field holds the list of marks of the respective subject.
- Is this enough? What happens when we call PhT.average()?
  - PhT.average() checks if PhT.aValue is -1.
  - If not, it just returns aValue
  - Otherwise, it computes the average from the marks in the marksList - which is just the average of the Physics marks !

Compare parametrised procedure with encapsulation

- Procedure Avemarks takes a field as a parameter
  - We can call Avemarks with Total or the subject name
- Datatype ClassAve has a procedure average() within it
  - We call average() for each object of ClassAve datatype
  - Advantage: result is stored, next call is much faster
  - Disadvantage: objects need to be created and initialised
- Caller is unaware of what is happening inside

AveTotal = Avemarks(Total)

AveMaths = Avemarks(Maths)

AvePhysics = Avemarks(Physics)

AveChemistry = Avemarks(Chemistry)

AveTotal = CT.average()

AveMaths = MaT.average()

AvePhysics = PhT.average()

AveChemistry = ChT.average()

But what if we the dataset is not static?

- If the dataset changes (consider for example that we add a new student to the class), then the stored average values will be wrong ! How do we deal with this?
- We need to manage the addition of a new student carefully
  - write a procedure addStudent(newMark) in the ClassAve datatype which takes as parameter the marks of the new student (total, or the respective subject marks)
  - A new student can be added only via the use of this procedure
- addStudent will need to append newMark at the end of its marksList
- addStudent will also need to reset aValue to -1, so that the average will get computed again

Using derived datatypes

- While the generic procedures count(), min(), max() and average() worked for all the categories, there could be procedures that work only for some categories and don't work for others
  - number() works for shirts but not for grapes
  - quantity() works for grapes, but may not work for shirts
- We can write derived datatypes to deal with this
  - Create derived types NumberCategory and QuantityCategory of Category
  - All the generic procedures count(), min(), max() and average() are available for objects of these derived types also
- We could then add more specific procedures to each derived type that will be available only to objects of that derived type
  - number() procedure is defined in NumberCategory
  - quantity() procedure is defined in QuantityCategory

Computational Thinking

Week 11

▼ **Concurrency**

- In all the procedures we have seen so far, only one step was being executed at a time
- In real life, several things are going on at the same time
- This means that two or more steps could be executed simultaneously this is called concurrency
- Specically, two procedure calls P() and Q() can be executed concurrently
- Concurrency means that the procedure calls have to be unbundled.

▼ The step B = P(A) involves the following different actions:

- Start the procedure P and pass the parameter A to it
- Wait for the procedure to complete its work.
- Accept the result of the procedure and store the result in B

▼ Easier to visualise concurrency with objects.

- For objects X and Y: X.P(A) and Y.P(A) may be executed concurrently
- In this lecture, we will only look at concurrency within object-oriented computing

▼ **Concurrent Objects**

▼ Step B = X.P(A) involves the following different actions:

- Start the procedure X.P and pass the parameter A to it
- We can write this as X.start(P,A)

- Object X is told to start its procedure P with parameter A

▼ Wait for the procedure X.P to complete its work

- Object X can indicate completion by setting a flag which can be read using X.ready(P)
- Object X can be polled by repeatedly checking if the condition X.ready(P) is true
- Meanwhile, function wait(C) can put the caller in a wait state till some condition C is true
- Putting it all together: wait(X.ready(P)) makes the caller wait for the result of P to be ready

▼ Accept the result of the procedure and store the result in B

- We can write this as B = X.result(P)

Example: Classroom dataset

- Consider two objects MaT and PhT of the ClassAve datatype
  - We wish to find MaT.average() and PhT.average() concurrently
- Recall that ClassAve had the following:
  - private fields marksList and aValue
  - public procedures average() and addStudent(newMark)
- To make average concurrent:
  - We need to implement the procedures ready(average) and result(average)
  - But we already have the field aValue which does this !
  - So ready(average) can just return the boolean value (aValue ≥ 0)
  - Similarly, result(average) can just return aValue

Example: Classroom dataset

- So to execute MaT.average() and PhT.average() concurrently:

```
MaT.start(average)
PhT.start(average)
wait(MaT.ready(average) and PhT.ready(average))
AveMaths = MaT.result(average)
AvePhysics = PhT.result(average)
```

- Note that:
  - start is called with only average, as it has no parameters
  - caller waits for both MaT and PhT to be ready

Example: Classroom dataset

- Now consider the same two objects MaT and PhT of the ClassAve datatype
  - We wish to add a new student to both MaT and PhT
  - MaT.addStudent(newMark1) and PhT.addStudent(newMark2) are executed concurrently
- To make addStudent concurrent:
  - We need to implement the procedure ready(newStudent)
  - We could use the same field aValue to implement this !
  - ready(addStudent) can just return the boolean value (aValue == -1)
  - Note that addStudent does not return anything, so we don't need the result implementation

Example: Classroom dataset

- So to execute MaT.addStudent(newMark1) and PhT.addStudent(newMark2) concurrently:

  MaT.start(addStudent, newMark1)
  PhT.start(addStudent, newMark2)
  wait(MaT.ready(addStudent) and PhT.ready(addStudent))

- Note that:
  - start is called with parameters
  - caller waits for both MaT and PhT to be ready

Example: Classroom dataset

- Now consider the situation where while MaT.average() is executing, a new student is added to MaT:

  MaT.start(average)
  MaT.start(addStudent,newMark)
  wait(MaT.ready(average) and MaT.ready(addStudent))
  AveMaths = MaT.result(average)

- There is potential for conflict here !
  - average will set aValue to 0 or a positive value
  - addStudent will set aValue to -1
  - So both cannot be ready at the same time !
  - wait(MaT.ready(average) and MaT.ready(addStudent)) will just wait forever !!
- To prevent this, we can use explicit aveReady and addReady fields:
  - addReady is set to false when addStudent starts and to true when it finishes
  - ready(addStudent) returns the value of addReady
  - Similarly for aveReady which is set after average completes

Race conditions

- addStudent needs to set two fields after it finishes
  - aValue has to be set to -1, so that the average is recomputed
  - addReady has to be set to true
- Similarly, average needs to set two fields after it finishes
  - aValue has to be set to the computed average value
  - aveReady has to be set to true
- While in the addStudent case, the two values can be written in any order, in the average case, aValue must be updated before aveReady is set
  - Otherwise, we could have a race condition: aveReady allows the wait condition to be true, so the wait exits
  - caller could then execute result(average) before the aValue is set: this will result in -1 being returned as the average !

Need for atomicity

- Concurrent execution of average and addStudent:

```
MaT.start(average)
MaT.start(addStudent,newMark)
wait(MaT.ready(average) and MaT.ready(addStudent))
AveMaths = MaT.result(average)
```

- What is the average returned? Does the average include the new student?
  - If average completes processing the list before the new student is added, then the average does not include the new student marks
  - On the other hand, if the new student has been added before average starts processing the list, then the average will include the new student marks
  - But something worse can happen ! What if average is at the end of the list, just as the new element is being appended to the end of the list?
  - May lead to a erroneous list being created, or may crash !

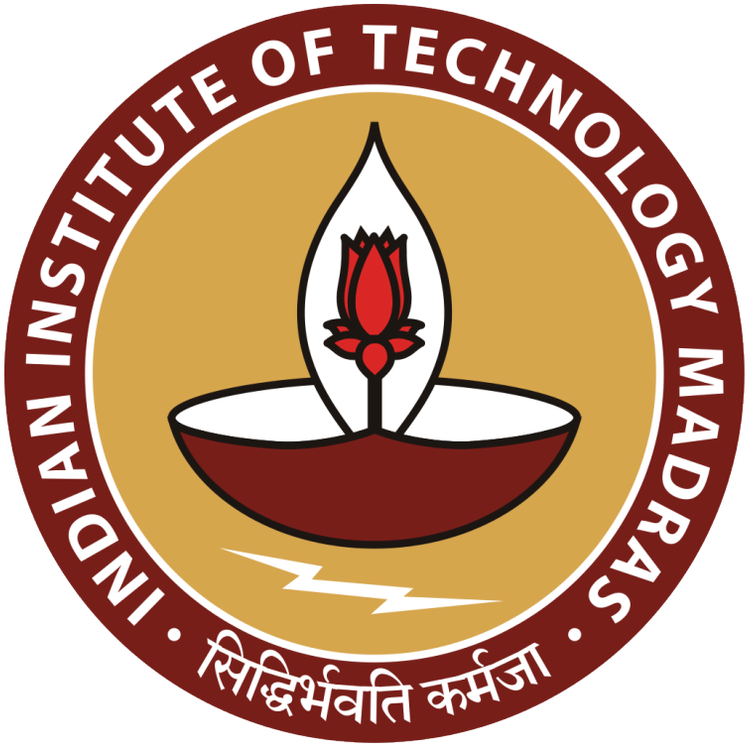
Need for atomicity

- We could argue: why do we need to execute average and addStudent concurrently. Just do them in sequence. For instance:

```
MaT.start(addStudent,newMark)
wait(MaT.ready(addStudent))
MaT.start(average)
wait(MaT.ready(average))
AveMaths = MaT.result(average)
```

- Will clearly return the average including the new student.
- The issue is while designing the concurrent object of datatype ClassAve, we cannot control who will call the procedures and in what order
  - One caller X may call addStudent, and a different caller Y may call average concurrently
  - It is very difficult to have X and Y co-ordinate on their use of the shared object MaT

- Concurrency allows several steps to be executed simultaneously
  - To keep control of this, we only considered concurrent execution of procedures within objects
- In the polling model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - The object can be polled to check if it has finished. The result can then be retrieved from the object (we used ready and result for this).
- In the producer-consumer model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases:
  - caller used wait(C) to wait for the boolean condition C to become true
  - In our polling model, C used ready() to check if the object has finished its task
  - In our producer-consumer model, C called available(B) to check if B has been fully written
- In both cases, race conditions have to be handled by ensuring that access to shared objects (such as lists) are made atomic (i.e. non-concurrent).



IIT Madras

ONLINE DEGREE

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

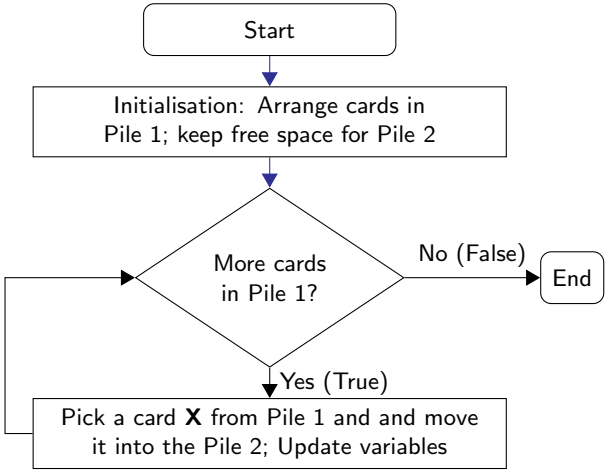
while (Pile 1 has more cards) {

  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

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
- **Depth first search** is a systematic procedure to explore a graph
  - Recursively visit all unexplored neighbours
  - Keep track of visited vertices in a dictionary
  - Can discover properties of the graph — for instance, is it connected?

Encapsulation

- **Encapsulation** packages procedures with the data elements on which they operate
  - Fields (or procedures) can be hidden from the outside world by marking them as **private**
  - Procedures act as the interfaces to the external world

Encapsulation

- **Encapsulation** packages procedures with the data elements on which they operate
  - Fields (or procedures) can be hidden from the outside world by marking them as **private**
  - Procedures act as the interfaces to the external world
- Encapsulation provides more modularisation than procedures
  - State is retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural
  - **Derived types** extend the functionality to specific instances
  - Disadvantage: objects need to be created and initialised at the start

Encapsulation

- **Encapsulation** packages procedures with the data elements on which they operate
  - Fields (or procedures) can be hidden from the outside world by marking them as **private**
  - Procedures act as the interfaces to the external world
- Encapsulation provides more modularisation than procedures
  - State is retained between calls ... can be used to speed up the procedures
  - Side effects are made explicit and more natural
  - **Derived types** extend the functionality to specific instances
  - Disadvantage: objects need to be created and initialised at the start
- "Object oriented computing" patterns can be found between different examples

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously
- This requires the **remote procedure call** to be unbundled. Two models:
  - In the **polling** model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - In the **producer-consumer** model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously
- This requires the **remote procedure call** to be unbundled. Two models:
  - In the **polling** model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - In the **producer-consumer** model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases, **race** conditions have to be handled by ensuring that access to shared objects (such as lists) are made **atomic** (i.e. non-concurrent).

Concurrency

- **Concurrency** allows several tasks to be executed simultaneously
- This requires the **remote procedure call** to be unbundled. Two models:
  - In the **polling** model, the caller starts all the procedures to be executed concurrently, and then waits for them to complete
  - In the **producer-consumer** model, the caller object starts all the procedures to be executed concurrently, and passes result buffers to them to write into when they finish
- In both cases, **race** conditions have to be handled by ensuring that access to shared objects (such as lists) are made **atomic** (i.e. non-concurrent).
- Concurrency can model Input/Output between users and a computer

Bottom-up computing

- In **bottom-up computing**, the code is constructed from (a sample of) the data elements
- In **classification**, a tree like structure (decision tree) is created
- **Prediction** tries to find a (linear) numerical function that connects the value to be predicted to the numerical values of the data elements that are available
- Classification and Prediction can be combined. Decision trees can use prediction functions with cutoffs, and prediction functions can be made specific to branches of a decision tree.

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