6, 2020

Week 1

DATASETS (Examples : Mark sheets, shopping Bills etc.)

Iteration : Going through a sequence of objects and performing the same operations on each object. For ex : counting.

Variable : An entity whose value keeps changing as the computation goes on.

Filtering : To take out specific type of data from entire dataset.

The pattern of doing something repititively is called an ITERATOR.

Description of The Iterator

Initialisation Step : arrange all the cards in an "unseen" pile

Continue or Exit : If there are no more cards in the "unseen" pile, we exit otherwise we continue.

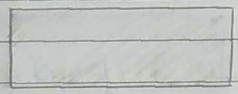
Repeat Step : pick an element from the "unseen" pile, do whatever we want to with this element, and then move it to another "seen" pile.

Go Back to Step 2.

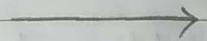
Date
October 6, 2020

INTRODUCTION TO Flowcharts

Some Commonly Used Symbols

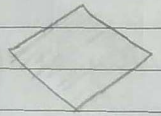


→ Process or Activity (set of operations that change the value of data (variables))



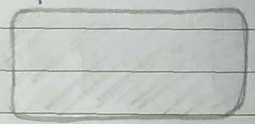
→ Flowline or Arrow (shows the order of execution of program steps)

→ Decision



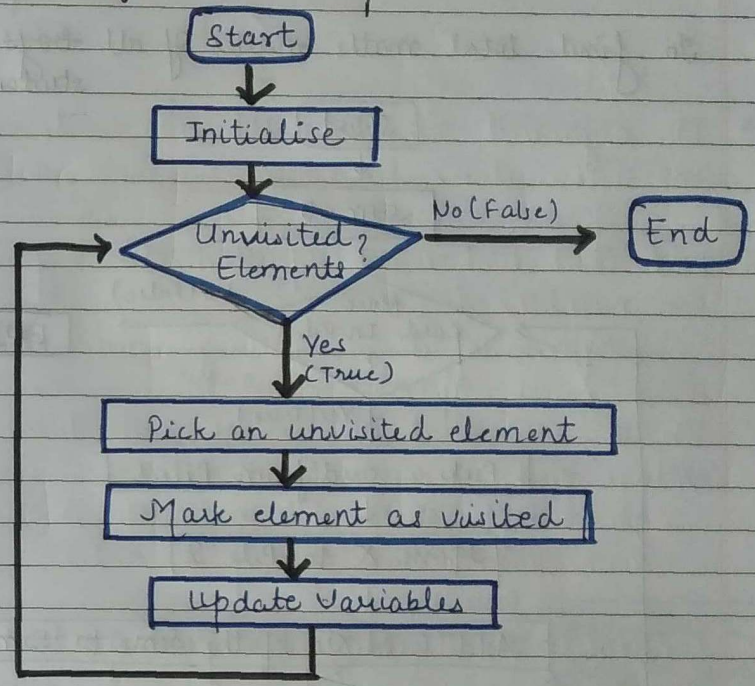
(Determines which path the program will take (condition))

→ Terminal



(Indicates the "Start" or "End" of the program)

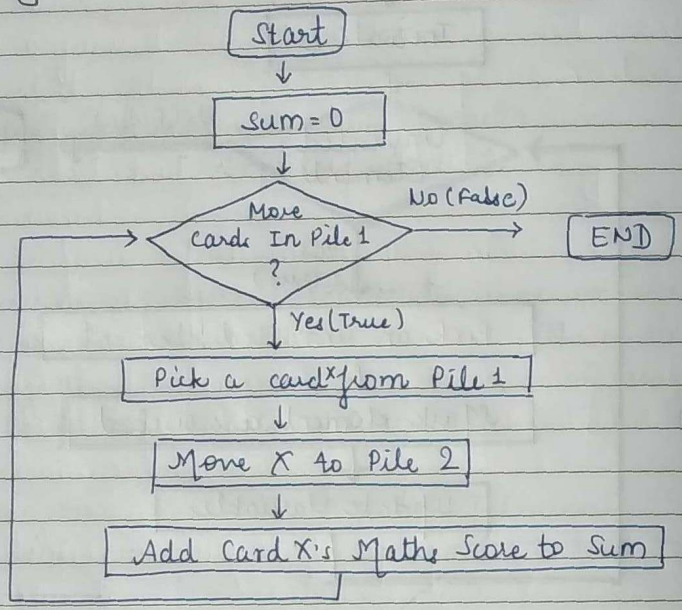
Generic flowchart for Iteration



Date
October 7, 2020

Some Flowchart Examples

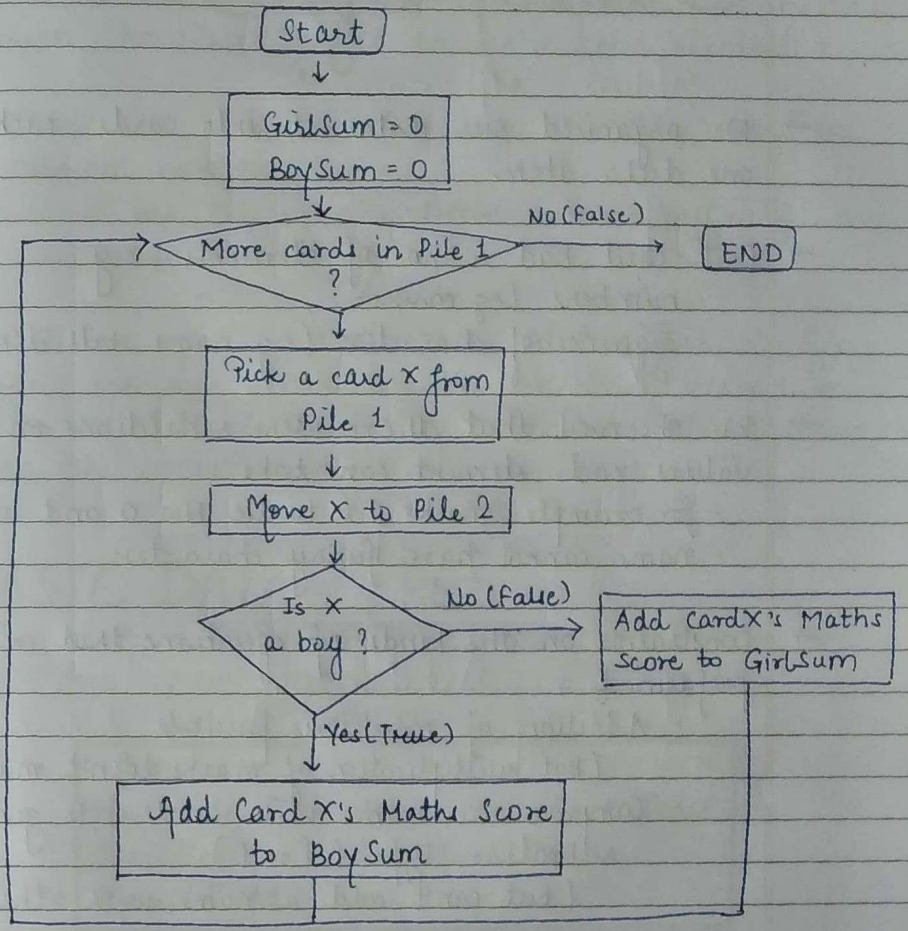
To find total maths marks of all ~~boys~~ from m.sheet. students



Modification

- If we only need boys marks
- If we only need girls marks
- If we need boys & girls marks separately

Sum of both Boys and Girls Maths marks



Nitu
October 7, 2020

Sanity of Data

→ We organised our data set into cards, each storing one data item

→ Each card had a no. of elements, e.g:
- numbers (eg 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 b/w 0 and 100
- name cannot have funny characters

→ Constraints on the kinds of operations that can be performed:
- Addition of marks is possible
  (but multiplication of marks doesn't make sense!)
- Compare one name with another to generate a boolean type (T or F)
  (but can't add a name with other)

This leads us to the concept of DATA TYPE...

→ By associating a 'DATA TYPE' (or simply type) with a data element, we can tell the computer (or another person) how we intend to use a data element:
- What are the values (or range of values) that the element can take?
- What are the operations that can be performed on the data element?

→ When we specify that a variable is of a specific type, we are describing the constraints placed on that variable in terms of the values it can store, and the operations that are permitted on it

BASIC DATA TYPES

Boolean

Has only Two Values : True or False
Operation : And, OR, NOT
Result Type : Boolean

Integer

Range of values is : ... -3, -2, -1, 0, 1, 2, 3...
Operation : +, -, x, ÷ ; <, >, =
Result Type : Integer ; Boolean

Character

Values - alphanumeric :
A B C ... Z a b .... z 0 1 2 9

Special Characters :
. , ; : * / & % $ # @ ! .....

Operation : =
Result Type : Boolean

String

Values - any sequence of characters
Operation : char in string ? ; =
Result Type : Boolean ; Boolean

Date January 8, 2021.

WEEK 3

PRESENTATION OF DATASETS IN THE FORM OF A TABLE

* Extracting Data From Cards

→ Each card is a unit of information
→ Diff. attributes and fields
  · Card Id, Name, Gender, ... Total
→ Organise as a table
→ All the grade cards in a single table

Summary

· Data on cards can be naturally represented using tables
· Each attribute is a column in the table
· Each card is a row in the table
· Difficulty if the cards has a variable no. of attributes
  → Items in shopping bills
  → Multiple rows - Duplication of data
  → Split as separate tables - need to link via unique attribute

PROCEDURE (Functions)

· A procedure to Sum up Maths marks

PSEUDOCODE

```
Procedure SumMaths (gen)
  Sum = 0
  while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    if (X.Gender == gen) {
      Sum = Sum + X.Maths
    }
  }
  return (sum)
end SumMaths
```

Procedure name : SumMaths
Argument recieves value : gen
Call procedure with a parameter SumMaths (F)
Argument variable is assigned parameter value.
Procedure call SumMaths (F), implicitly starts with gen = F
Procedure return the value stored in sum

· A procedure to sum up any type of marks

```
Procedure SumMarks (gen, fld)
  Sum = 0
  while (Pile 1 has more cards) {
```

```
Pick a card X from Pile 1
Move X to Pile 2
if (X.Gender == gen) {
    Sum = Sum + X.fld
  }
}
return (Sum)
end SumMarks
```

Two parameters, gender (gen) and field (fld)
gen is assigned a value, M or F, to check against X.Gender
fld is assigned a field name, to extract appropriate card entry X.fld
Single Procedure SumMarks to handle diff. requirements
  - SumMarks (F, Chemistry)
    - Sum of Girls Chemistry marks
  - SumMarks (M, Physics)
    - Sum of Boys physics marks

• Calling A Procedure

Use procedure name like a math function, as part of an expression.
Assign the return value to a variable

```
GirlChemSum = SumMarks (F, Chemistry)
BoyChemSum = SumMarks (M, Chemistry)
  if (GirlChemSum > BoyChemSum) {
      "Congratulate the Girls"
    }
  else
    { "Congratulate the Boys" }
```

A procedure may not return a value
Correct marks for one subject on a card
  - Procedure UpdateMarks (CardId, Sub, Marks)
Procedure call is a separate statement
  - UpdateMarks (17, Physics, 88)

SUMMARY

- Procedures are pseudocode templates that work in different situations
- Delegate work by calling a procedure with appropriate parameters
  - Parameter can be a value, or a field name
  - SumMarks (M, Total)
- Calling a procedure
  - Procedure call is an expression, assign return value to a variable
    - GirlChemMarks = SumMarks (F, Chemistry)
  - No useful return value, procedure call is a separate statement
    - UpdateMarks (17, Physics, 88)

- Procedures help to modularise pseudocode
  - Avoid describing the same process repeatedly
  - If we improve the code in a procedure, benefit automatically applies to all procedure calls

Example : Analysis of Top Students

- Is there a single student who is the best performer across subjects?
- Is the highest overall total the same as the sum of the highest marks in each subject?
- Need to compute maximum for diff. fields in a score card
  - Maths, Physics, Chemistry, Total
- Ideally suited to using procedures
  - same computation with a parameter to indicate the field of interest

* Find the maximum in a given field

```
Procedure MaxMarks (fld)
  Maxval = 0
  while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    if (X.fld > Maxval) {
        Maxval = X.fld
    }
  }
  Return (Maxval)
end MaxMarks
```

- As usual keep track of maximum using a variable
  - initialise to 0
  - update whenever you see a bigger value
- The value to be compared is not fixed
  - Parameter fld determines the field of interest

```
MaxMaths = MaxMarks (Maths)
MaxPhysics = MaxMarks (Physics)
MaxChem = MaxMarks (Chemistry)
MaxTotal = MaxMarks (Total)
SubjTotal = MaxMaths + MaxPhysics + MaxChem
if (MaxTotal == SubjTotal) {
    SingleTopper = True
}
else {
    SingleTopper = False
}
```

- Use the procedure MaxMarks to find maximum marks in different categories
  - Four procedure calls, with fld set appropriately
  - Save each return value separately
- Use saved return values to compare the maximum overall total with the sum of the maximum subject totals.

SIDE EFFECTS OF PROCEDURES

- Side Effect → Procedure modifies some data during its computation.
- Sequence of cards may be disturbed
- Does it matter?
  → not in this case - adding marks does not depend on how the cards are arranged
- Sometimes the side effect is the end goal
  → procedure to arrange cards in decreasing order of Total Marks.
- A side effect could be undesirable
  → We pass a deck arranged in decreasing order of Total Marks
  → After the procedure, the deck is randomly rearranged

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

Summary.

- Need to separate interface & implementation
- Interface describes a contract
  · Parameters to be passed
  · Value to be returned
  · What side effects are possible
- Can change the implementation provided we preserve the interface.
- Side effects are important to be aware of
  · Sometimes no guarantee is needed (adding up marks)
  · Sometimes no side effect is tolerated (pronoun matching)
  · Sometimes the side effect is the goal (sort the data)

Date
January 9, 2021

Week 4

BINNING

Importance of Binning to reduce no. of comparisons in nested iterations.

- REDUCING COMPARISONS - What we observed!

⇒ Some computations seem to require comparisons of each card with all the other cards in the pile
→ for example, choosing a study partner for each student
→ the no. of comparisons required can be very large

⇒ We observed that if we can organize the cards into bins based on some heuristic:
→ then we only need to compare cards within one bin
→ this seems to significantly reduce the no. of comparisons req.

⇒ Is there a formal way of determining the reduction in comparisons?
→ Calculate the no. of comparisons w/o binning
→ Calculate the no. of comparisons with binning
→ Use these calculations to determine the reduction factor

Comparing Each Element With All Other Elements

For 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)

No. of comparisons = 4+3+2+1 = 10

⇒ For N objects, the no. of comparisons required will be:
- (N-1) + (N-2) + ... + 1
- which is = $\frac{N(N-1)}{2}$

⇒ This is same as the no. of ways of choosing 2 objects from N objects:
- $NC_2 = \frac{N \times (N-1)}{2}$

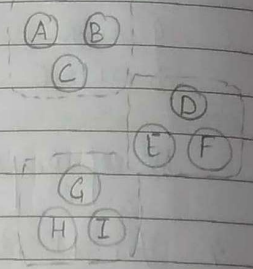
⇒ From first principles:
- Total no. of pairs is N×N
- From this reduce self comparisons (e.g. A with A). So no. is reduced to: (N×N) − N
  which can be written as N(N-1)
- Comparing A with B is same as comparing B with A, so we are double counting this comparison
- So reduce the count by half = $\frac{N(N-1)}{2}$

∴ No. of Comparisons (w/o Binning) = $\frac{N(N-1)}{2}$

Disadvantage: the no. of comparisons grows really fast !!

How do we reduce the no. of comparisons?

KEY IDEA : Use Binning



- For 9 objects A, B, C, D, E, F, G, H, I:
  → The no. of comparisons
  $= \frac{9(9-1)}{2} = \frac{9 \times 8}{2} = 36$

- If the objects can be binned into 3 bins of 3 each:
  → The no. of comparisons per bin is : $\frac{3(3-1)}{2} = \frac{3 \times 2}{2} = 3$
  → Total no. of comparisons for all 3 bins = $3 \times 3 = 9$

- So the no. of comparisons reduces from 36 to 9!
  → Reduced by a factor of 4 times.

Calculation of Reduction Due To Binning

For N items:
No. of comparisons w/o binning = $\frac{N(N-1)}{2}$

If we use K bins of equal size, no. of items in each bin is $\left(\frac{N}{K}\right)$

∴ No. of comparisons per bin = $\frac{\left(\frac{N}{K}\right)\left(\frac{N}{K}-1\right)}{2}$

⇒ Total no. of comparisons = $K\left(\frac{\frac{N}{K}\left(\frac{N}{K}-1\right)}{2}\right)$

$= \frac{N}{2}\left(\frac{N}{K}-1\right)$

Thus, Factors of Red^n is : $\frac{1}{2}N(N-1) \div \frac{1}{2}N\left(\frac{N}{K}-1\right)$

Factor of Reduction = $\frac{(N-1)}{\left(\frac{N}{K}-1\right)}$

∴ For N = 9 and K = 3, this is $\left(\frac{9-1}{9/3-1}\right) = \frac{8}{3-1} = \frac{8}{2} = 4$

So, reduction is by a factor of 4 times.

SUMMARY

* The no. of comparisons b/w all pairs of items grows quadratically, i.e, quite fast
* The formula of no. of comparisons for N items is : $N(N-1)/2$
* Sometimes, it is possible to find a heuristic that allows us to put the items into bins and compare only items within the bins.
* If there are N items put into K bins of equal size, then the no. of comparisons reduces to $\frac{N}{2}\left(\frac{N}{K}-1\right)$
* The factor red^n is $\frac{(N-1)}{\left(\frac{N}{K}-1\right)}$.

Summary Of All 4 Weeks

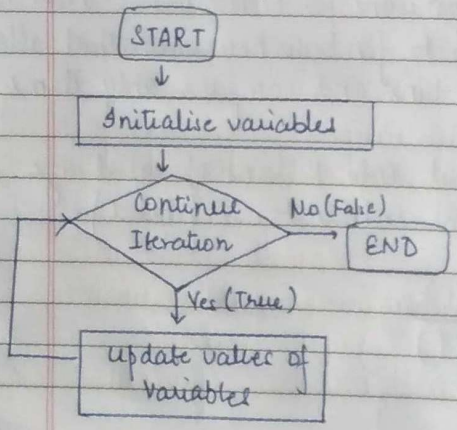
Iterators And Variables

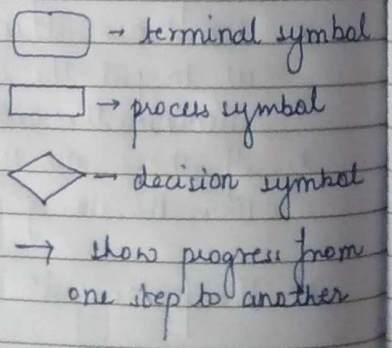
- The iterator is the most commonly used pattern in CT.
- Represents the procedure of doing some task repeatedly
  - → required an initialisation step
  - → the steps for the task that needs to be repeated
  - → A way to determine when to stop the iteration

- Variables keep track of intermediate values during the iteration.
  - → Variables are given starting values at the intialisation step
  - → At each repeated step, the variable values are updated

- Initialisation and update of variables are done through assignment statements.

Iterator represented as a flowchart





Iteration expressed through pseudocode

```
Initialise variables
  while (continue with iteration) {
      update values of variables
  }
```

Iteration to systematically go through a set of items

```
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

- Sanity of different data field of the item leads us to the concept of DATATYPES, which clearly identifies the values and allowed operations
- Basic data types : boolean, integer, character
- Add to this string data type
- 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 same data type
- Whereas a record has multiple named fields, each can be of a different datatype

Iteration with Filtering

- Filtering makes a decision at each repeated step whether to process an item or not.
- This introduces a decision step within the iteration loop
- Expressed in pseudocode, it would look something like this:

```
Initialise variables
while (continue with iteration?) {
    ....
    if (condition is satisfied?) {
        update some variables
    }
    ....
}
prepare final results from variable values
```

- The filtering condition can compare the item values with a constant
  ⇒ The filtering condition does not change after each iteration step (is constant)
  Example; Count, Sum
- Or it could compare item values with a variable
  ⇒ The filtering condition changes after an iteration step
  Example: max

Procedures & 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 variable
- Instead of writing the code again with a small difference, we now just have to make a call to the procedure with a different parameter value.
- E.g. finding max for each subject.

Accumulation through Iteration

- The most common use of an iterator is to create an aggregate value (accumulation) from the available values.
- Simple examples of this are count, sum, average
- We could also apply filtering while doing accumulation eg. sum of boys' marks.

- We could also collect a list of elements
  e.g. list of students with max marks in a subject.

Doing Two Iterations - one after another

- Use the first iteration to do some accumulation
- The variables in which these accumulations are done can be called accumulators.
- Second iteration can do filtering using the accumulator variables.
- E.g. finding above avg. students - avg. is an accumulator
- This establishes a relationship b/w any element and the aggregate of all elements
  e.g. find out the more frequently occuring word.

Doing Two Iterations - one nested within another

- If we need to go beyond the relationship b/w an element and the aggregate of all elements
  ... to expressing a relationship b/w any 2 elements

→ We will need to do one iteration within another
  eg. find out if 2 students have same birth day

→ Nested Iterations are costly in terms of no. of computations required.

→ We could use the no. of comparisons by using binning wherever possible.

Date January 10, 2021

WEEK 5

Pseudocode : Introducing Lists

COLLECTIONS

- Variables keep track of intermediate values

- Often we need to keep track of a collection of values
  → 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]
  → ["Ram", "cane", "Monday"]
  → [ ] : empty list

- Append two lists, l1 ++ l2
  - → l1 is [1,13] and l2 is [2,7,1]
  - → l1 ++ l2 is [1,13,2,7,1]

- Extend l with item x
  - → l = l ++ [x]

- Examples
  - → List of students born in May
  - → List of students from Chennai

CODE ★

```
chennailist=[ ]
while (Table 1 has more rows) {
  Read the first row X in Table 1
  if ( X.TownCity == "Chennai") {
     chennailist = chennailist ++ [X.segno.]
  }
  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

CODE ★

```
maychennailist = [ ]
  foreach x in maylist {
      foreach y in chennailist {
          if ( x == y ) {
              maychennailist = maychennailist ++ [x]
          }
      }
  }
```

SUMMARY

- A list is a sequence of values
- Write a list as [x1, x2, x3, ..., xn]
- Combine lists using ++
  - → [x1, x2] ++ [y1, y2, y3] → [x1, x2, y1, y2, y3]
- Extending list l by an item x
  - → l = l ++ [x]
- foreach iterates through values in a list

```
foreach x in l {
    Do something with x
}
```

Date
January 11, 2021

Lists

PSEUDOCODE for operations on the data collected in 3 prizes problem using lists

- 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 3 lists
  · Obtain cutoffs in each subject
  · Initialise lists for each subject
  · Scan each row
  · For each subject, check if the marks are within top 3
  · If so, append to the list for that subject
  · First find the students who are toppers in Maths & Phy
  · Then match these toppers with toppers in chem.

CODE

```
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 > secondmax) {
      thirdmax = secondmax
      secondmax = X.subj
    }
    if ( secondmax > X.subj > thirdmax) {
      thirdmax = X.subj
    }
    Move X to Table 2
  }
  return (thirdmax)
End TopThreeMarks

cutoff Maths   = TopThreeMarks(maths)
cutoff Physics = TopThreeMarks(Physics)
cutoff Chemistry = TopThreeMarks(Chemistry)

mathsList = []
phyList = []
chemList = []

while (Table 1 has more rows) {
  Read the first row X in Table 1
  if ( X.Maths >= cutoff Maths) {
    mathsList = mathsList ++ [X.seqno]
  }
}
```

```
    if (X.Physics >= ~~phyli~~ cutoffPhysics) {
        phyList = phyList ++ [X.SeqNo]
    }
    if (X.Chemistry >= cutoffChemistry) {
        chemList = chemList ++ [X.SeqNo]
    }
  Move X to Table 2
}

mathsPhyList = [ ]
mathsPhyChemList = [ ]

for each x is mathsList {
    for each y in phyList {
        if (x == y) {
            mathsphyList = mathsphyList ++ [x]
        }
    }
}

for each x in mathsphyList {
    foreach y in chemList {
        if (x == y) {
            mathsphychemList = mathsphychemList ++ [x]
        }
    }
}
```

SUMMARY

- Lists are useful to collect items that share some property
- Nested Iteration can find common elements across two lists
- Can group lists to process more than 2 lists
  - → Find common items across four lists, l1, l2, l3, l4
  - → Nested iteration on l1, l2 constructs l12 of common items in first two lists
  - → Nested iteration on l3, l4 constructs l34 of common items in last two lists
  - → Nested iteration on l12, l34 finds common items across all four lists

Sorting Lists

Arranging Lists In Order

- Sorting a list often makes further computations simple
  - → Finding the top k values
  - → Finding duplicates
  - → Grouping by percentiles - top quarter, next quarter...
- Many clever algorithms exist, we look at a simple one
- 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

CODE :

```
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 its empty
  - → Inserting into a sorted list

CODE :

```
Procedure InsertionSort (l)
    sortedList = [ ]

    foreach z in l {
        sortedList = SortedListInsert (sortedList, z)
    }
    return (sortedList)
End InsertionSort
```

CORRELATING MARKS IN MATHS AND PHYSICS

- We want to test the following hypothesis
"Student who performs well in Maths performs at least as well in Physics"

- 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 g.

- 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 and B list for Physics
  - → Use these counts to confirm or reject hypothesis

ASSIGNING GRADES

- Assign grades {A, B, C, D} approximately at quartile boundaries
  - → Top 25% get A, next 25% get B, next 25% get C, bottom 25% get D.

- To calculate quartiles, extract marks as a list and sort the list

- Need to identify the students - each entry in the marks list is a pair [StudentId, Marks]

- Procedure to extract marks information as a list for a subject

- Get the marks lists for Maths and Physics

- Use Insertion Sort for mathList and phyList ?
  - → Entries are [id, marks]
  - → To compare [i1, m1] and [i2, m2], only look at m1, m2

- Extracting values at the beginning and end of a list
  - → first(l) and last(l)
    first([1,2,3,4]) is 1, last([1,2,3,4]) is 4
  - → The remainder of the list is given by rest(l) and init(l), respectively
    rest([1,2,3,4]) is [2,3,4],
    init([1,2,3,4]) is [1,2,3]

- Modify SortedListInsert

- InsertionSort uses updated SortedList

- Assign grades to a sorted list by quartile
  - → length(l) returns number of elements in l
  - → compute quartile boundaries based on class size
  - → Initialise list for each grade
  - → Assign grades based on the position in the list

- SimpleGradeAssignment returns a list containing 4 lists, for the 4 grades

- Assign grades corresponding to Maths & Physics marks
- Unpack the four lists into 4 separate lists

Test the hypothesis

- Check how many students with A in Maths confirm the hypothesis
  → exitloop prematurely terminates a foreach loop
- Check how many students with B in Maths confirm the hypothesis
- Finally, check length(confirm) against length(confirm) + length(reject) to decides if the hypothesis holds.

CODE :

```
Procedure BuildMarksList (field)
  marksList = [ ]
  while (Table 1 has more rows) {
    Read the first row X in Table 1
    marksList = marksList ++ [[X.segno, X.field]]
    More X to Table 2
  }
  return (marksList)
End BuildMarksList

mathsList = BuildMarksList (Maths)
phyList = BuildMarksList (Physics)
```

```
Procedure SortedListInsert (l, x)
  newlist = [ ]
  inserted = false

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

  if ( not (inserted)) {
      newlist = newlist ++ [x]
    }
  return (newlist)
End SortedListInsert

sortedMathsList = Insertionsort (MathsList)
sortedPhyList = Insertionsort (PhyList)

Procedure SimpleGradeAssignment (l)
  classSize = length(l)
  q4 = classSize / 4
  q3 = classSize / 2
  q2 = 3 * ClassSize / 4
  gradeA = [ ]
  gradeB = [ ]
  gradeC = [ ]
```

```
gradeD = [ ]

position = 0
foreach x in l {
    if (position > q2) {
        gradeA = gradeA ++ [ first(x) ]
    }
    if ( position > q3 and position <= q2) {
        gradeB = gradeB ++ [ first (x) ]
    }
    if ( position > q4 and position <= q3) {
        gradeC = gradeC ++ [ first (x) ]
    }
    if ( position <= q4) {
        gradeD = gradeD ++ [ first(x) ]
    }
    position = position + 1
}
return ( [gradeA , gradeB, gradeC , gradeD ] )
End SimpleGradeAssignment

mathsGrades = SimpleGradeAssignment (sortedMathsList)
physicsGrades = SimpleGradeAssignment ( sortedPhyList)

mathsAGrades = first (mathsGrades)
mathsBGrades = first ( rest (mathsGrades) )
mathsCGrades = last ( init (mathsGrades) )
mathsDGrades = last ( mathsGrades )

phyAGrades = first ( phyGrades )
phyBGrades = first ( rest (phyGrades) )
phyCGrades = ( init ( phyGrades) )
phyDGrades = ( last ( phyGrades) )

confirm = [ ]
reject = [ ]

foreach x in mathsAGrades {
    found = False
    foreach y in phyAGrades {
        if ( x == y ) {
            confirm = confirm ++ [x]
            found = True
            exitloop
        }
    }
    if (not (found)) {
        reject = reject ++ [x]
    }
}

// foreach x in MathsB //
foreach x in MathsBGrades {
    found = false
    foreach y in PhyAGrades {
        if ( x == y ) {
            confirm = confirm ++ [x]
            found = True, exitloop
        }
    }
}
if (not (found)) {
    foreach y in PhyBGrades {
```

```
if (x == y) {
    confirm = confirm ++ [x]
    found = true
    exit loop
  }
 }
}

if ( not (found)) {
   reject = reject ++ [x]
  }
}
```

SUMMARY

- Sorting was used to identify quartiles for graded assignment
- Need to modify the comparison fn based on the items in the list
- length [l] returns no. of elements in l
- New functions to extract first & last items of a list
  - first (l) and rest (l)
  - init (l) and last (l)
- exit loop to abort a foreach loop

length (l) to find length of list (no of elements)

List Functions

- length (l)
- first (l)
- last (l)
- rest (l)
- init (l)
- member (l, e)

(1) length (l) Eg. length ([8, 9, 3]) is 3

```
length (l) {
      count = 0
      foreach x in l {
          count = count + 1
      }
    return (count)
}
```

(2) first (l) Eg. first ([20, 30, 40, 50]) is 20

```
first (l) {
    foreach x in l {
        return (x)
    }
}
```

(3) last (l) Eg last ([1, 8, 9, 2]) is 2

```
last (l) {
```

```
    foreach x in l {
        e = x
    }
    return (e)
}
```

(4) rest (l) Eg. rest ([1,2,3,4]) is [2,3,4]

```
rest (l) {
    found = False
    restList = [ ]
    foreach x in l {
        if (found) {
            restList = restList ++ [x]
        }
        else {
            found = True
        }
    }
    return (restList)
}
```

(5) init (l) Eg. init ([1,2,3,4]) is [1,2,3]

```
init (l) {
    found = False
    initList = [ ]
    foreach x in l {
        if (found) {
            initList = initList ++ [prev]
        }
        else {
            found = True
        }
        prev = x
    }
    return (initList)
}
```

(6) member (l, e) Eg. member ([20, 30, 40], 30) is True ;
member ([20, 30, 40], 10) is False

```
member (l, e) {
    foreach x in l {
        if (e == x) {
            return (True)
        }
    }
    return (False)
}
```

Date January 17, 2021

WEEK 6

Introducing Dictionaries

Indexed Collections

- 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 pair. For instance
  - → Chemistry marks (value) for each student (key)
  - → Source station (value) for a train route (key)
- Present the key to extract the value - takes the same time for all keys, random access
  - → m = chemMarks["Rahul"]
  - → s = sourcestation["10211"]

Pseudocode for Dictionaries

- At a 'raw' level, sequence of key:value pairs within braces
  - → {"Rahul": 92, "Clarence": 73, "Ritika": 89}
- Empty dictionary : { }
- Access value by providing key within square brackets
  - → s = sourcestation["10215"]
- Assigning a value - replace value or create new key-value pair
  - → chemMarks["Rahul"] = 92
- Dictionary must exist to create new entry
  - → Initialise 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 Avg. 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

- Typical use of a dictionary is to accumulate values
  → runs["Kohli"], runs scored by Virat Kohli
- Process a dataset with runs from different matches
- Each times we see an entry for Kohli, update his score
  → runs["Kohli"] = runs["Kohli"] + score
- What about the first score for Kohli?
  → Create a new key and assign score
  → runs["Kohli"] = score
- How do we know whether to create a fresh key or update an existing 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
}
```

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

- Implementing isKey(d, k)
  → iterate through keys(d) searching for the key k
- Takes time proportional to size of the dictionary
- Instead, assume isKey(d, k) is given to us, works in constant time
  → Random Access

Summary

- A dictionary stores a collection of key : value pairs
- Random access - getting the value for any key takes constant time
- Dictionary is sequence
  $\{k_1 : v_1, k2 : v2, k3 : v3, \ldots, kn : vn\}$
- Usually, create an empty dictionary and add key-value pairs

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

- Iterate through a dictionary using keys(d) → list

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

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

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

Dictionary : Real Time Ex.

Customers Buying Food Items

- Find the customer who buys the highest amount of food items
- Create a dictionary to store food purchases
  - → Customer names as keys
  - → No. of food items purchased as values

CODE

```
foodD = { }
while (Table1 has more rows) {
    Read the first Row X in Table1
        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 Table2
}
```

Birthday Paradox

- Find a birthday shared by more than 1 student
- Creates 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 dob as a dictionary

CODE →

```
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.
- Iterate through the dictionary to match pronouns
- Note that partofspeech["Noun"] and partofspeech["Pronoun"] are both sorted in ascending order of SerialNo

CODE →

```
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
}

matchD = { }
```

```
foreach p in partofspeech ["Pronoun"] {
    matched = -1
    foreach n in partofspeech ["Noun"] {
        if (n < p) {
            matched = n
        }
        else {
            exitloop
        }
    }
    match[p] = matched
}
```

Side Effects of Dictionary

Dealing with side effects

- Comparing two lists for duplicate items:
  - Nested Loop

→
```
Procedure FindOverLap (l1, l2)
    overlap = [ ]
    foreach x in l1 {
        foreach y in l2 {
            if (x == y) {
                overlap = overlap ++ [x]
            }
        }
    }
    return (overlap)
End FindOverLap
```

- What if the lists are sorted?
  - Need not start inner iteration from beginning
  - Use first() & rest() to cut down the list to be scanned

→
```
Procedure FindOverLap2(l1, l2)
    overlap = [ ]
    foreach x in l1 {
        y = first(l2)
        l2 = rest(l2)
        while (y < x) {
            y = first(l2)
            l2 = rest(l2)
        }
        if (x == y) {
            overlap = overlap ++ [x]
        }
    }
    return (overlap)
End FindOverLap2
```

x ↓

l1 = [1, 2, 6, 8, 9]

l2 = [2, 5, 8, 11]

overlap = [2, 8]

- 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

CODE :

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

- 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

Summary :

- Be careful of side effects when working with collections
  - → Make a local copy of the argument
- Sometimes, side effects are convinient for updating collections in place
  - → Deleting a key in a dictionary
  - → Sorting list
- Can also return a new collection and reassign after the procedure call
  - → myd = DeleteKey2 (myd, k)
  - → mylist = InsertionSort (mylist)