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.

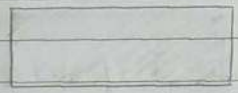
Repeat Step : pick an element from the "unseen" pile, do whatever we want to with this element, and then move it to another "seen" pile.

Go Back to Step 2.

Date
October 6, 2020

INTRODUCTION TO Flowcharts

Some Commonly Used Symbols



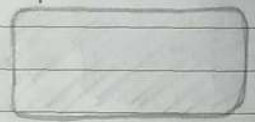
→ Process or Activity (set of operations that change the value of data (variables))

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

→ Decision

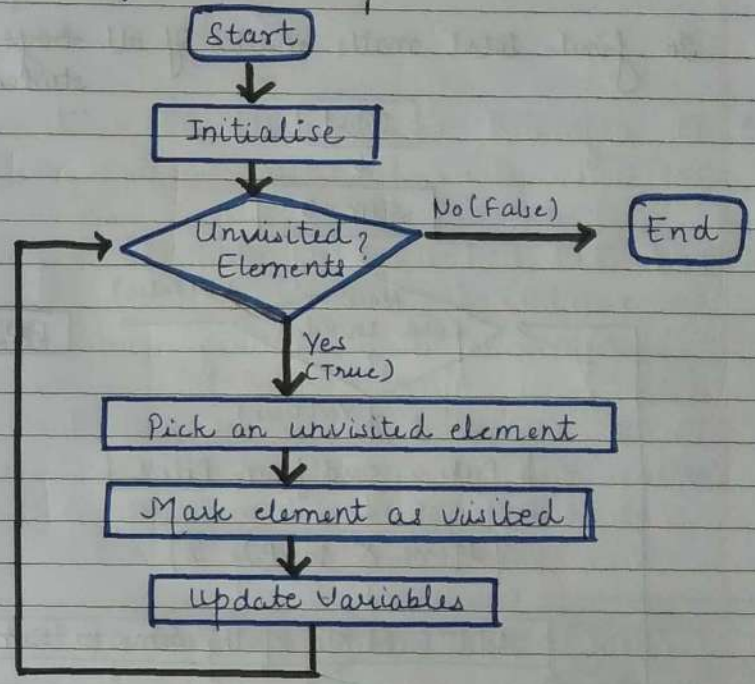
(Determines which path the program will take {condition})

→ Terminal



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

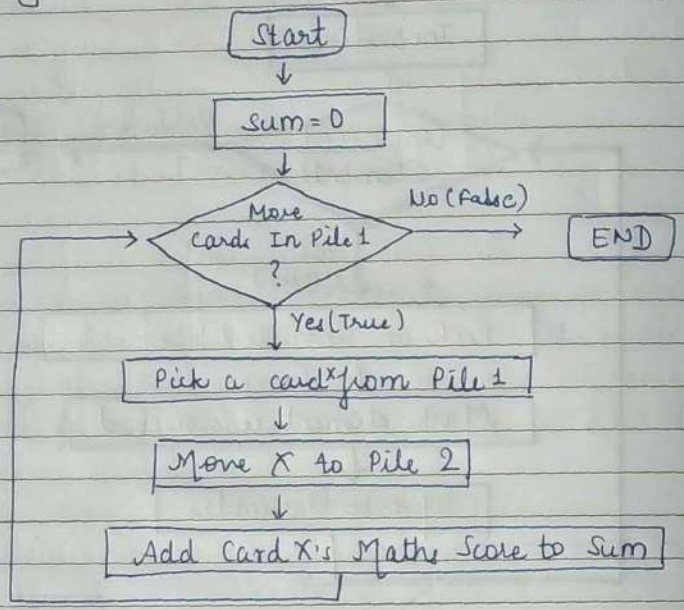
Generic flowchart for Iteration



Date
October 7, 2020

Some Flowchart Examples

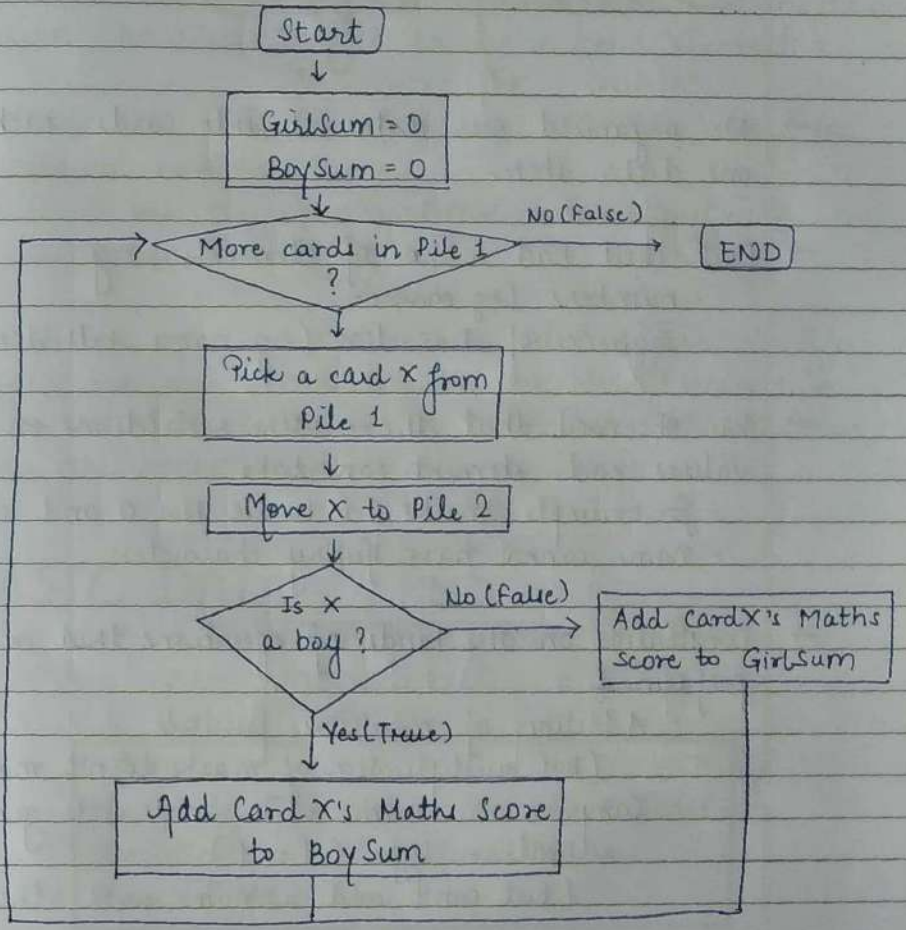
To find total maths marks of all ~~boys~~ from m.sheet. students



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



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:
- $(N-1) + (N-2) + \dots + 1$
- 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 $N \times N$
- From this reduce self comparisons (e.g. A with A). So no. is reduced to: $(N \times N) - N$
  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 = $K\left(\frac{N/K\ (N/K - 1)}{2}\right)$

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

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

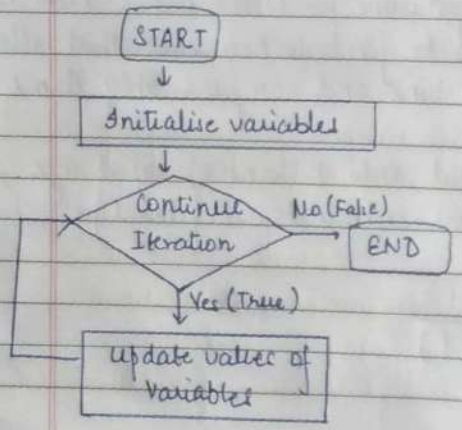
Iterators And Variables

- 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 updates of variables are done through assignment statements.

Iterator represented as a flowchart



→ terminal symbol

→ process symbol

→ decision symbol

→ show progress from one step to another

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

cutoffMaths   = TopThreeMarks(maths)
cutoffPhysics = TopThreeMarks(Physics)
cutoffChemistry = TopThreeMarks(Chemistry)

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

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

```
    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
    marks list = markslist ++ [[X.segno, X.field]]
    More X to Table 2
  }
  return (marks list)
End BuildMarks List

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)
  class Size = 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 mathsA Grades {
    found = False
    foreach y in phyA Grades {
        if (x == y) {
            confirm = confirm ++ [x]
            found = True
            exitloop
        }
    }
    if (not (found)) {
        reject = reject ++ [x]
    }
}

// foreach x in MathsB //
foreach x in MathsB Grades {
    found = false
    foreach y in PhysA Grades {
        if (x == y) {
            confirm = confirm ++ [x]
            found = True, exitloop
        }
    }
    if (not (found)) {
        foreach y in PhysB grades {
```

```
        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) E.g. 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) E.g. 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
forcach 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
  $\{k_1 : v_1, k_2 : v_2, k_3 : v_3, \ldots, k_n : v_n\}$
- 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 (is key (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)

Date January 28, 2021

WEEK 7

Introducing Matrices

COLLECTIONS

- A list keeps a sequence of values
  - → No random access
  - → For value at position i, start at the beginning and scan i-1 elements
- A dictionary stores key-value pairs
  - → supports random access
  - → keys can be arbitrary values
- Often we need a matrix
  - → Two dimensional table
  - → m rows, n columns
  - → Random access to matrix[i][j]
    - → By convention, rows & columns are numbered from 0
    - → $0 \leq i \leq m-1$, $0 \leq j \leq n-1$

IMPLEMENTING MATRICES

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

```
my matrix = CreateMatrix (30, 45)
```

CODE →

```
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

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

We can also change the order by iterating columns first & then rows!!

- Iterating through the rows
  - Rows 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

CODE →

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

We can also process matrix columnwise :

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

SUMMARY

→ 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 & column wise
  - foreach r in rows(mymatrix)
  - foreach c in columns(mymatrix)

→ Matrices will be useful to represent graphs

Working with Graphs

MENTORING

- Student A can mentor student B in a subject if A has higher marks, but not too much higher,
  → Diff is b/w 10 & 20 marks
- Create a dictionary for marks in subject,
  → maths marks = ReadMarks (Mathematics)
- Creating 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

CODE →

```
Procedure ReadMarks (subj)
  marks = { }
  while (Table 1 has more rows) {
    Read the first row X in Table 1
    marks [X.SeqNo] = X.subj
  }
  Move X to Table 2
  return (marks)
End ReadMarks

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
```

PAIRING STUDENTS IN STUDY GRPS

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

CODE →

```
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 (ikey (paired, i)) and not (ikey (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 collect them.

CODE →

```
mathmarks = ReadMarks (Mathematics)
phymarks = ReadMarks (Physics)
chemmarks = ReadMarks (Chemistry)

mathMentorGraph = CreateMentorGraph (mathMarks)
phyMentorGraph = CreateMentorGraph (~~chem~~Marks)
                                    phy
chemMentorGraph = CreateMentorGraph (chemMarks)

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

CODE →

```
Procedure CreateSimilarityGraph ( mark1, mark2, mark3)
  n = length ( keys(mark1))
  similarityGraph = CreateMatrix(n,n)

foreach i in keys (mark1) {
    foreach j in keys (mark1) {
        ijDiff1 = abs( mark1[i] - mark1[j] )
        ijDiff2 = abs( mark2[i] - mark2[j] )
        ijDiff3 = abs( mark3[i] - mark3[j] )

        If ( ijDiff1 ≤ 10 and ijDiff2 ≤ 10 and ijDiff3 ≤ 10) {
            similarityGraph[i][j] = 1
        }
    }
}
return ( similarityGraph)
End CreateSimilarityGraph
```

SUMMARY

- Graphs are 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 info. from the graph

Date
February 3, 2021

WEEK 8

Graphs - Trains

TRAINS

- Train dataset - information about trains and stations
  → Each train is a route list of stations
  → Each station is a route list of trains passing through
- Compute pairs of stations connected by a direct train
- Represent start and end station of trains in a nested dictionary.
  → trains[t][start] , trains[t][end]

DIRECT ROUTES

- Stations A & B such that a train starts at A and ends at B.
- First, compile the list of stations from 'trains'.
- Create a matrix to record direct routes
- Map station ~~map~~ names to row and column indices.
- Populate the matrix

CODE →

```
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)
    strindex = { }
    i = 0
    foreach s in keys (stations) {
        strindex [s] = i
        i = i + 1
    }
    foreach t in keys (trains) {
        i = strindex [trains [t] [start]]
        j = strindex [train [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
- Iterate through intermediate stations
  → set onehop [i] [j] = 1, if there is a connection via intermediate station k.
- More useful to let onehop [i] [j] mean "connected with at most one hop"
  → initalise onehop to include direct routes

CODE →

```
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
- Extend a one hop route from i to k by a direct train from k to j
- Iterate through intermediate stations
  → Combine information in direct and onehop

→ Check onehop [i][k] and direct [k][j]

CODE →

```
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 nhops record connections from station A to station B by changing at most n trains
- Want to extend nhops to allow one more hop
- Iterate through intermediate stations
- Combine information direct and nhops
  → Extend an nhop route from i to k by a direct train from k to j

→ Check nhops [i][k] and direct [k][j]

CODE →

```
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
```

DISCOVERING PATHS

- Path : sequence of edges from A to B
  → A direct edge is a path of length 1
- Procedure OneMoreHop extends paths of length n to paths of length n+1
- N nodes – shortest path from A to B has at most N-1 edges
  → a longer path would visit a node twice
  – unnecessary loop

- Transitive Closure – pairs of nodes connected by a path
  → Repeat OneMoreHop N-1 times starting with direct

SUMMARY

- 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
- Compute the transitive closure of edge relation
  → stop with paths of length N-1 for an N node graph

Date
February 4, 2021

Edge Labelled Graphs

ADDING INFO. TO DIRECT ROUTE GRA.

- Stations A and B such that a train starts at A & ends at B
- 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
- Information about trains is recorded as a label on the edge
  → Edge labelled graph

CODE →

```
Procedure LabelledDirectRoutes (trains)
⋮
foreach r in rows (direct) {
    foreach c in 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 LabelledDirectRoutes
```

DISTANCES B/W 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 b/w same pair of stations
- Graph directdist
  → Edge label : directdist[i][j] is the shortest distance b/w i & j.
- 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.

CODE →

```
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
- Initialise 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.

CODE →

```
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
```

SUMMARY

- We can represent extra info. in a graph via edge labels
  - Train name, distance
  - Foreach edge in the graph, replace 1 in matrix by edge label
- Iteratively update labels
  - Compute shortest distance path b/w each pair of stations

February 6, 2021

WEEK 9

ADA 2 → Q4

Recursion

INDUCTIVE DEFINITIONS

- Many computations are naturally defined inductively
  - Base case: directly return the value
  - Inductive step: compute value in terms of smaller arguments
- Factorial
  - $n! = n \times (n-1) \times \ldots \times 2 \times 1$
  - $0!$ is defined to be 1
  - factorial (0) = 1
  - For $n > 0$, factorial (n) = n * factorial (n-1)

CODE →

```
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)

CODE :
```
Procedure ListSum (l)
  if ( l == [] ) {
    return (0)
  }
  else {
    return ( first(l) + ListSum(rest(l)) )
  }
End ListSum
```

→ Can also add last(l) to sum of init(l)

CODE :
```
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.

CODE :
```
Procedure InsertionSort (l)
  sortedList = []
  foreach z in l {
    sorted_list = SortedListInsert ( sortedList , z)
  }
  return ( sortedList)
End InsertionSort

Procedure SortedListInsert ( l , z)
  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 SortedInsertList
```

INSERTION SORT, INDUCTIVELY

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

CODE :

```
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
```

SUMMARY

- Many functions are naturally defined in an inductive manner
  - → Base case and inductive step
  - → For numeric functions, base case is typically 0 or 1
  - → For lists, base case is typically length 0 or length 1
- Use recursive procedures to compute such functions
  - → Base case : value is explicitly calculated & returned
  - → Inductive case : value requires procedure to evaluated on a smaller input
  - → Suspend the current computation till the recursive computation terminates
- WARNING : Without properly defined base cases, recursive procedures will not terminate.

A mutual recursion is a form of recursion in which two objects are defined in terms of each other.

Recursion is a process in which function calls itself until termination condition is not true.

It requires a base case and an inductive step to complete the process.

It makes tree traversal easier.

Depth-First Search

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

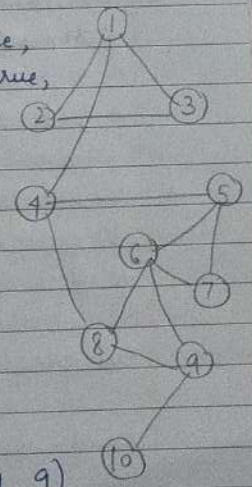
DEPTH FIRST SEARCH

- Maintain info about visited notes in a dictionary visited
- Recursively update visited each time we explore an unvisited neighbour
- To explore vertices reachable from i
  - → Initialise 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.

CODE:

```
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: output:
visited = {4: True, 1: True, 2: True, 3: True, 5: True, 6: True, 7: True, 8: True, 9: True, 10: True}



code:
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)

SUMMARY

- Depth First Search is a systematic procedure to explore a graph
- Recursively visit all unexplored graphs
- Keep track of visited vertices in a dictionary
- Can discover properties of the graph - for instance, is it connected?

DFS : 1) It is a recursive algorith
2) It uses the idea of 'backtracking'
3) It explores all possible searches at each node before backtracking.

* A standard DFS implementation puts each vertex of the graph into one of the two categories : visited & not visited
* A DFS tree don't contain cycles
* In DFS technique if all searches exhaust at one of the node, it goes back to 'just previous node'.

Date February 8, 2021
MONDAY

WEEK 10
Encapsulation

WHAT IS 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 & 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 "how?".

DO WE NEED ANYTHING BEYOND PROCEDURES ?

- Procedures already provide some kind of encapsulation
  - → Interface via the 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
  - → 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)

DATA TYPES & ENCAPSULATION

- Recall datatypes?
  - → Basic datatypes : integer, character, boolean
  - → Subtype of a datatype to restrict the values & 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 datatypes
- Operations on a datatype
  - → Well defined operations on the basic datatypes & their subtypes
  - → Some operations on the string datatype
- But what about operations on compound datatypes?
- What if we 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 :

(1) Classroom Scores Dataset

- Questions more frequently asked of the scores dataset
  - → What is the avg in a subject - say Maths?
  - → What is the overall avg?
- Suppose that we've to ask these questions many times?
  - → It is wasteful to carry out same computation again & again.
  - → Can we not store the answer of a question, & just return the saved answer when the question is asked again?
- This will work as long as the data is static.

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 marklist 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.

- What about the avg values of the individual subjects?

- Just like CT, we could have objects PhT, MaT and ChT (for Phy, Maths & Chem. 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 & ChT need only hold the Maths & Chemistry marks lists.

- So let us say that each of these objects are also of the same datatype ClassAvg, but their marks list 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 marks list - which is just the average of the Physics marks!

- So the same datatype classAvg can be used for all the objects, CT, PhT, ChT and MaT.

COMPARE PARAMETERISED PROCEDURE with ENCAPSULATION

- Procedure AvgMarks takes a field as a parameter
  - → We can call Avgmarks with Total or subject name

```
AvgTotal = Avgmarks(Total)
AvgMaths = Avgmarks(Maths)
AvgPhysics = Avgmarks(Physics)
AvgChemistry = Avgmarks(Chem)
```

- Datatype ClassAvg has a procedure average() within it
  - → We call average() for each object of ClassAvg datatype
  - → Advantage: result is stored, next call is much faster
  - → Disadvantage: object needs to be created & initialised

```
AvgTotal = CT.average()
AvgMaths = MaT.average()
AvgPhysics = PhT.average()
AvgChemistry = ChT.average()
```

- Caller is unaware of what is happening inside.

BUT WHAT IF WE THE DATASET IS NOT STATIC?

- If the dataset changes (consider for example that we add a new student to 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 ClassAvg datatype which takes as parameter the marks of the new student (total, or 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.

COMPLETE EXAMPLE

- So what does ClassAvg datatype look like at the end of all this?

- Has two data fields and two procedures
  → field marksList contains a list of marks
  → field aValue which is either -1 (not computed) or holds the computed average value.
  → procedure average() returns aValue if it is not -1, else it computes the avg of marksList, stores it in aValue and returns it.
  → procedure addStudent(newMark) appends newMark to the end of marksList and resets aValue to -1.

- Note that the procedures 'average' and 'addStudent' are assumed to have access to the fields marksList and aValue, so we can reference these fields by just using their name (it is as if they are implicitly passed as parameters to the procedure)

- Note also that the procedures are allowed to (and expected to!) make changes to these fields - so the potential side-effects are made explicit

- But what if the external world access marksList or aValue directly?

- ClassAvg should be able to restrict access to its fields

- This is typically done by declaring the field as either a private or a public field.

- ClassAvg has two private fields and two procedures:
  → private field marksList contains a list of marks
  → private field aValue which is either -1 (not computed) or holds the computed average value.
  → public procedure average() returns aValue if it is not -1, else it computes the avg of marksList & stores it in aValue and returns it.
  → public procedure addStudent(newMark) appends newMark to the end of marksList & resets aValue to -1

- Note that a procedure could also be declared private in which case it is not available to outside world.

(2) Shopping Bills DataSet

- Questions asked of the shopping bill dataset
  - → How many items of a specific category (e.g. shirts) were sold?
  - → What is the minimum, maximum and avg. unit price for that category?
- We could define a datatype Category and create an object Shirts of this datatype
- What are its fields?
  - → copy of the entire shopping bill dataset can be stored but this is wasteful
  - → We only need the list of items (i.e rows of the bills) which are of that category
  - → Category could have a private field itemList which carries all the row items from all the bills which are all of the same category
- The datatype could also encapsulate procedures that provide the desired information:
  - → count() returns the no. of items (i.e size of the list)
  - → min() returns the least value of unit price at which the category item was sold
  - → max() returns the largest value of the unit price
  - → average() returns the average across all unit prices for that category.
- The Category datatype encapsulates one private field and four procedures
  - → private field itemList carries the list of items, which is hidden
  - → public procedures count(), min(), max() & average() that anyone can call
- What if we want to do this for a particular shop? Or for a specific customer?
  - → The itemList can store only the items sold by that shop
  - → ... or by a specific customer
  - → The procedures will all work without any change!
- Multiple objects of type Category can be created - the list being filtered by category alone, or by a combination of category and customer name and shop name, or anything else
- What if we want to accelerate the execution of these procedures?
  - → Just store their results in private fields - say cValue for count, minValue for min, maxValue for max and aValue for average.
- Can you see how the pattern for encapsulating the Category is quite similar to that for encapsulating the ClassAve?
- If Shirts is an object of this datatype, we may want to find out how many shirts were sold?

→ Note that this may not be the same as count(), since one row in the bill can have multiple items
→ We can write a procedure called number() which finds the sum of the quantity field of the list items

- But number() makes sense only if the category is sold in discrete units (like shirts). What if the category quantity is measured in Kg (for eg. grapes)?

→ For such categories, we can define a procedure quantity that finds the sum of the (possibly fractional) quantity fields.

- The issue is that just like number() does not make sense for grapes, a measure of quantity in Kg does not make sense for shirts!

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 & QtyCategory of Category.
→ All the generic procedures count(), min(), max() & average() are available for objects of that derived datatypes 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 QtyCategory

SUMMARY

- Encapsulation allows procedures to be packaged together with the data elements on which they operate.

→ This is achieved by allowing procedures in addition to fields in a datatype
→ Procedures are the interfaces for others to interact with the objects of this datatype
→ The procedures can have side effects in terms of modifying the field values of the object
→ Fields (or procedures) can be hidden from the outside world by making them as private

- Encapsulation provides an increased level of modularisation when compared to procedures

→ Allows state to be retained b/w calls ... can be used

to speed up the procedures.

→ Side effects are made explicit & more natural

→ We can use derived types to extend the functionality to specific instances when needed

→ But the disadvantage is that objects needs to be created & initialised at the start

- We can find common "object oriented computing" patterns between diff-examples.