Date
January 28, 2024

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.regno] = 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