Pseudocode: 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

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

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 and columns are numbered from 0
    - $0 \leq i \leq m - 1$, $0 \leq j \leq n - 1$

Implementing matrices

- Dictionaries support random access

Implementing matrices

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

```
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
```

Implementing matrices

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

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

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

Processing matrices

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

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

Processing matrices

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

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

Processing matrices

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

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

- Iterating through the rows
  - Row indices are keys of outer dictionary
  - Column indices are keys of first (any) row

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

Processing matrices

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

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

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

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

Processing matrices

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

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

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

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

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

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

Processing matrices

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

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

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

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

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

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

Can also process a matrix columnwise

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 and column-wise
  - `foreach r in rows(mymatrix)`
  - `foreach c in columns(mymatrix)`
- Matrices will be useful to represent graphs