

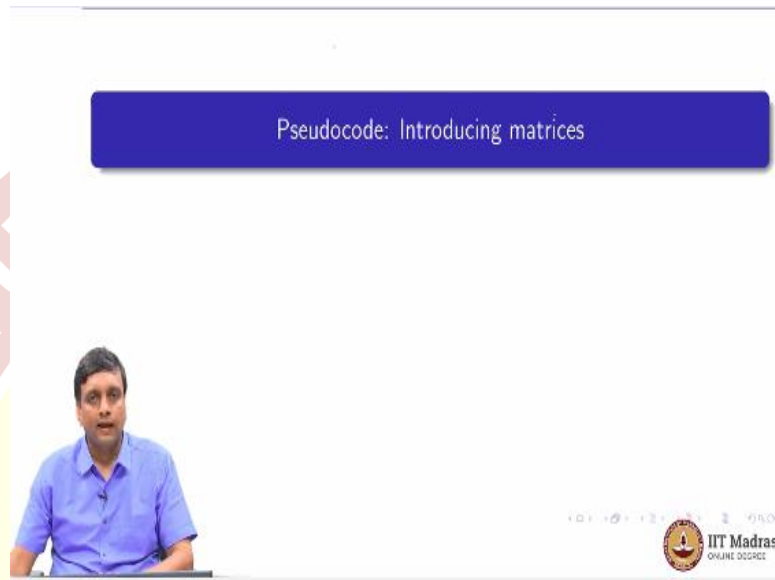


IIT Madras

ONLINE DEGREE

Computational Thinking
Professor G.Venkatesh
Department of Computer Science
Indian Institute of Technology, Madras
Introduction to matrices and implementation of matrix using nested dictionary

(Refer Slide Time: 00:14)



So we have seen two kinds of collection so far lists and dictionaries.

(Refer Slide Time: 00:20)

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
 - $0 \leq i \leq m-1, 0 \leq j \leq n-1$

Handwritten notes: matrix[i][j], matrix[i,j]

Pseudocode: Introducing matrices 2/5 IIT Madras ONLINE DEGREE

So a list keeps a sequence of values. It is very useful for accumulating things as we go along, but one of the disadvantages of using a list is that it does not support what is called random access. So we cannot look at an arbitrarily element at the list without walking to it from the beginning. So, if you want a value in a list which is at a position i then you have no option except to start with the 0th position and walk right until you hit the i th position.

So this is a list then we said that if we really want random access we will assume that we have another kind of a collection called a dictionary. So in a dictionary we store values not by position, but by key. So a dictionary is a collection of key value pairs. The keys can be arbitrary values and the assumption is that whenever we want the value associated with the key it will take the same amount of time regardless of the size of the dictionary.

And which key we choose so this is what we called random access. It really does not depend on what you are looking for whereas in the list depending on the position that you are looking for it will take less or more time. So now we want the third type of collection which we call a matrix. So a matrix is what you would have seen in school it is just a two dimensional table with rows and columns. So each row is of the same length.

So in general for instance you may have m rows and n columns and the way that we will index these rows and columns is by using numbers from 0 to $m-1$ for the rows and 0 to $n-1$ for the column. So this is a convention that is often used in computing. Normally when you are using it in mathematics you will normally number them 1 to n and 1 to m , but it is very standard to number thing starting from 0 so we will just follow this convention.

So if you have m rows the rows are numbered 0 to $m-1$ and if we have n columns the columns are numbered 0 to $n-1$. So we will use the square bracket notation so if matrix is such a matrix so matrix is the name of a variable which stores such a matrix than to get the value at row i and column j we will use this double square bracket. So notice that this is a slightly different from maybe what you might intuitively think which is to use a pair of indices within a single square bracket.

So we will use this double square bracket which says that in matrix i the row i the j th element. So question now is how would we implement this? Do we need some new concepts in order to use matrices for our work?

(Refer Slide Time: 02:53)

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
```

The diagram shows a 2D array with rows and columns. The first row is labeled 'row' and the first column is labeled 'col'. The array is filled with zeros. The indices i and j are shown as arrows pointing to the first row and first column respectively.

Pseudocode: Introducing matrices 3/5

IIT Madras
ONLINE DEGREE

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
```



Pseudocode: Introducing matrices

3/5



So we already have a collection which supports random access namely a dictionary. So an easy way to implement matrices for our purpose is just to assume that they are special kinds of dictionaries. So, in particular we can think of a matrix as a nested dictionary. At the top level the keys corresponds to the rows so I have key 0 to m-1 which correspond to the rows and for each of these keys the value is another dictionary with keys 0 to n-1.

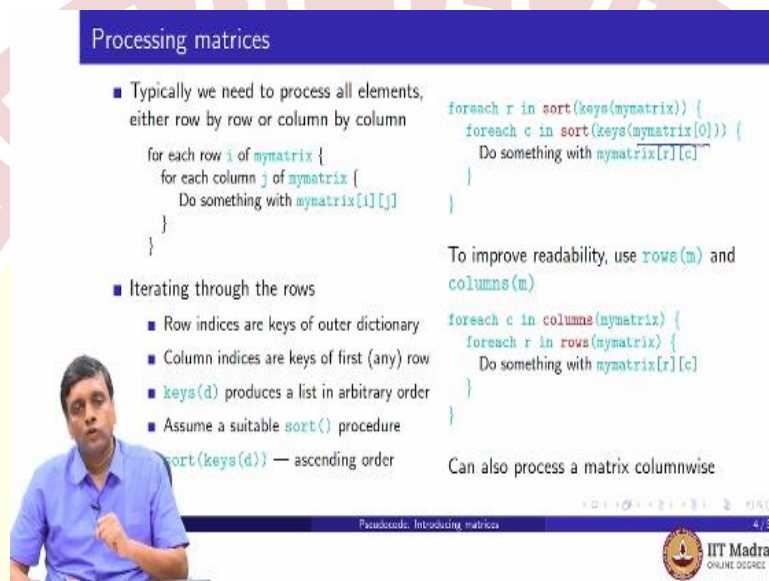
So here is how we would create a matrix with a given set of rows and a given set of columns. So we pass the argument how many rows I want and how many columns I want then I start with an empty dictionary which I have called mat and I start with the 0th row. So for each row I first create the empty dictionary that will store the row. So basically we are saying that each row is itself a dictionary.

So we have this dictionary mat which has key 0 to m-1 and then each of these is itself a dictionary which has key 0 to n-1. So the first thing I do for each row is I will create this dictionary empty dictionary corresponding to that row and now for each column that is for j from 0 to n-1 or in this case cols-1 since cols is the number of columns I want.

I will create this key and I will increment the column so that I will go to the next column and then I will stop and finally when I am done with this row I will increment the row. So this is just a straightforward nested loop again. So row by row I will create a dictionary for that row and for each row column by column I will create a key for that row and that column. So this is my procedure create matrix which can then be used in a matrix form.

Because now I can provide any argument ij and because these are just dictionary the ij th entry will take an equal amount of time by the property that we have assumed for dictionaries. So when I want to create a matrix I will just pass two numbers the number of rows I want and the number of columns I want and create matrix will create this nested dictionary and give it back to me and I assign it to be a variable like this. So my matrix is create matrix 30, 45 so now I have row indices 0 to 29 column indices 0 to 44.

(Refer Slide Time: 05:12)



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(m)` and `columns(m)`

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

Can also process a matrix columnwise

Paradocci: Introducing matrices 4/5 IIT Madras ONLINE COURSE

So the way we normally deal with matrices is to process them systematically either row by row or column by column. So, for instance we might want to say for each row in my matrix and for each column in my matrix do something with the element i, j at row i and column j . We could also do it the other way round we could also say for every column in my matrix and for every row in my matrix.

So basically instead of processing it row by row I am processing it column by column it depends on the application which one you want to do, but both of these are natural operations to perform on a matrix. So what we would like is an easy way to do this. So we want to have a way for example of iterating through the rows. Now remember that the row indices are just the keys of the outer dictionary.

Remember it is a nested dictionary so a matrix of i of j . So matrix of i those keys are actually the row indices and then within each matrix of i there are keys 0 to $n-1$ which are the column indices. So I can iterate through the row indices and the column indices so this would be one

way to do this. So for every r for every row in the keys of my matrix, for every column in the keys of the 0^{th} row of my matrix.

So remember all the rows have the same size so this is one assumption that this is a rectangular matrix. It does not matter whether I pick up 0 or 1 or any other row, but I know that if it is a non-empty matrix as at least one row and if that one row exist it will have index 0 so it is safe to pick up index 0. So for every row in the keys of my matrix and for every column in the keys of the first row of that matrix do something with the entry r, c at that point.

So this more or less corresponds to what we were intuitively saying over here, but there is a small problem and the small problem is that these keys need not come in the order that we expect. So when we said that keys of a dictionary returns the list of keys we did not specify that these keys come in any particular order. They may not even be in the order in which they were inserted and they are certainly not expected to be sorted in any particular order.

However, when we are processing a matrix it is natural to go from the first row to the last row and from the first column to the last column. So we would like these keys to come in that order 0 to $m-1$ for the rows, 0 to $n-1$ for the column. So that is easily done provided we have at our disposal. If you remember we wrote in one of our earlier lectures on list we wrote insertion.

So we could write a variety of such procedures. So let us assume we have a procedure called sort which will take a list and return an ascending order list in sorted order. So all now we need to do is to take this list of keys that is produced by the keys function and pass it through sort and now I am okay. If I say for every row in the sorted list of keys of my matrix and for every column in the sorted list of keys of the column of row 0 do something.

Then I am guaranteed that I am processing the rows in the correct sequence and the columns in the correct sequence. So this is an implementation that we could use, but because we are going to use it all the time it is a bit tedious to write the sort and keys and all that. So to improve readability we will just use rows and columns as a kind of shortcut to refer to those values sorted keys of my matrix.

So basically I am going to replace this complicated expression by this simpler expression. So rows of my matrix will always be sort of keys of my matrix. Columns of my matrix will

always be sort of keys of my matrix 0, but if I do it this way then when I am using a matrix it is very transparent what I am going to do I am going through the rows or I am going through the columns I am not lost in this notation of sorting and keys and all that.

So we will use this notation. So for each row and the rows of my matrix for each column and the columns of my matrix do something with my matrix at row, c and as before we can invert this now because this entry exist. So it is not that these entries have to be created. Remember that when we created a matrix we actually created every key. So we created every key and in fact we also initialize it to 0 this is the point I forgot to mention.

But create matrix creates every key i, j so there are no missing keys and every i, j entry is actually assigned a value 0 to start with because that is more or less how we are going to use it. So it is useful to combine the creation and the initialization in one step that is what we did. So my matrix 0 always exist so I can as well process the process before the rows there is no problem of whether my matrix 0 this value my matrix 0 is defined or not because it is defined.

So I can just say for c and columns for r and rows do something. So this is now a complete introduction to how we are going to use matrices. So we are going to use this create matrix we are always going to use only two dimensional matrices by the way. So we are going to pass two arguments the rows and the columns numbers to create a matrix and then once we have a matrix we can use this for each in rows and for each in columns in order to systematically process the matrix from beginning to end.

(Refer Slide Time: 10:28)

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(m)`
 - `foreach c in columns(m)`
- Matrices will be useful to represent graphs



Pseudocode: Introducing matrices



To summarize the matrix is just a two dimensional table and what we are basically going to exploit is that accessing any m, i, j value in this whole table takes the same amount of time. So these are random access collections and because they are random access collections and we do not want to create a new concept we have just implemented them for simplicity using dictionaries.

So these are actually dictionaries with special keys the keys are 0 to $m-1$ and 0 to $n-1$ and we have defined these again this is just a convenience we can write it in terms of the dictionary underlying it, but for convenience we have defined these new iterative like keys of d we have defined `rows` of m and `columns` of this should be m columns of m as new functions which will return us in sorted order the indices corresponding to the rows and the columns.

So the indices are always 0 to $row-1$ the last row or $n-1$ if there are m rows 0 to $n-1$ and this rows of m and columns of m will always give us the keys in that order. What we will see is that matrices are very useful to represent graphs which we have encountered in our lectures the way of representing relationships.