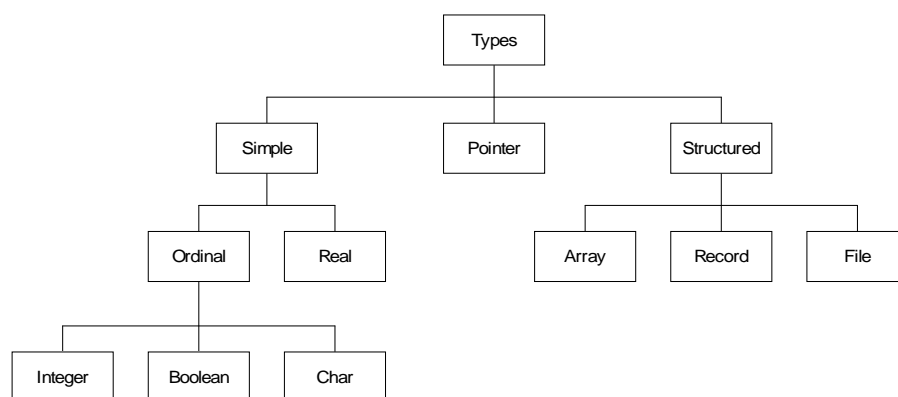


# Chapter 8

---

## □ DATA STRUCTURES

The data types in Pascal are categorized as follows :



**Figure 8.1.** Pascal Data Types.

We will NOT cover *enumerated*, *set* or *pointer* types.

### 8.1 TYPE DEFINITIONS

The most important feature that Pascal introduced to programming languages was the ability to ‘extend’ the language by *defining new types in terms of previously-defined types*. This is done by means of the **type definition**.

## Syntax of the Type Definition

```
TYPE
  <type-identifier> = <type-definition>;
  <type-identifier> = <type-definition>;
  etc.
```

**Examples** (Subrange types) These must refer to *ordinal* types only.

```
type
  PosIntegerType = 1..maxint;
  UpperCaseType  = 'A'..'Z';
  NonPosIntType  = - maxint..0;
  OneToTenType   = 1..10;
```

Once we have defined new types in a program we can use them as if they were part of the language.

```
program Types;
const
  MaxNo = 200;
  MinNo = 50;
type
  PosIntType = 1..maxint;
  RangeType = MinNo..MaxNo;
var
  i, j, k : RangeType;
  m       : PosIntType;
  n       : RangeType;
begin
  < statements>
end.
```

Why should we define new types and then declare variables with these new types when we could have declared the variables  $i, j, k, m, n$  as **integer** without having to define **consts** and **types**? One reason is to allow us to write **safer programs** that can be checked at *compile-time* or *run-time* to ensure that variables don't fall outside their defined ranges.

For example, the Lotto QuickPick is a program should generate pseudo-random integers in the range 1 . . 42. The following program does this.

```

program QuickPick;
  {$R+ This tells the compiler to generate range-checking code}
  const
    MinNo = 1;
    MaxNo = 42;
  type
    QPNoType = MinNo..MaxNo;
  var
    QPNumber : QPNoType;
    i        : 1..6;
  begin
    Randomize; {initialize random number generator}
    for i := 1 to 6 do begin
      QPNumber := Random(MaxNo) + 1;
      WriteLn(QPNumber);
    end; { FOR }
  end. { PROG QuickPick }

```

In the program above  $Random(MaxNo)$  is a built-in TurboPascal function that returns a random number between 1 and  $MaxNo$ .

**Exercise 8.1.1 :** What is wrong with the program above. Would you buy a *Quickpick* card from this program? □

## 8.2 STRUCTURED TYPES

The second reason for defining our own types is that this allows us to construct complex structured data types – data structures that all but the simplest programs use.

There are 4 built-in structured types in Pascal : Arrays, Records, Files, and Sets.

### 8.2.1 Arrays

An array is a group of elements , all of the same type—called the *element type* — and indexed by a set of values—called the *index type*. An array is similar to a vector in mathematics.

A	1	2	3	4	5	6	7	8	9	10	11	12
	22	14	11	6	74	24	12	81	31	10	53	7

**Figure 8.2.** An Array of Integers.

### Syntax of Array Type-Definition

```
type
  <Array type-identifier> = array[<index type>] of <element type>;
```

where <index type> is any previously-defined ordinal type  
and <element type> is any previously-defined type.

EXAMPLE : *Array Definitions*

```
type
  RArrNType = array[1..100] of real;
  CodeArrType = array['A'..'Z' ] of integer;
  TFArrType = array[-10..50] of boolean;
  CharArrType = array[-100..-5] of char;
var
  A : RArrNType;
  Code : CodeArrType;
```

□

Note that a **type** definition does not create (allocate) memory boxes; only **var** declarations create memory boxes according to the template specified by the variable's type.

**Why do we need arrays?** Suppose we have a team of 20 salesmen and each month we wish to print out a list of those salesmen that sell more than the average. We could do this as follows :

1. Read in monthly sales of each salesman.
2. Calculate the average sales.
3. Go through the sales again and print out those that are above the average.

The following program does these steps without using arrays. As we can see, the problem requires many line of code. Imagine if we had 1000 salesmen — we would need about 4000 lines of code for this very simple problem.

```
program SalesCalc1;
var
  Name1, Name2,..., Name20   : string;
  Sales1, Sales2,..., Sales20 : real;
  AvgSales                   : real;
begin
  Read(Name1); ReadLn(Sales1);
  Read(Name2); ReadLn(Sales2);
  .
  .
  .
  Read(Name20); ReadLn(Sales20);
  AvgSales := (Sales1 + Sales2 + ... + Sales20)/20;
  if( Sales1 > AvgSales) then begin
    WriteLn(Name1, Sales1);
  end;
  if( Sales2 > AvgSales) then begin
    WriteLn(Name2, Sales2);
  end;
  .
  .
  .
  if( Sales20 > AvgSales) then begin
    WriteLn(Name20, Sales20);
  end;
end. { PROG SalesCalc1 }
```

A much more elegant solution is to use two arrays : one for names and one for sales figures. We can do all the processing using **for** -loops with one or two statements per loop. Here is the same program using arrays and loops.

```

program SalesCalc2;
const
    MaxSMen = 2000;
type
    NameArrType = array[1 . . MaxSMen] of STRING;
    SalesArrType = array[1..MaxSMen] of real;
var
    Name : NameArrType;
    Sales : SalesArrType;
    i, NoSMen : 1..MaxSMen;
    AvgSales, sum : real;
begin
    Write('Type in number of salesmen (<= 2000) : ');
    ReadLn(NoSMen);
    for i := 1 TO NoSMen do begin
        Write('Name : '); ReadLn(Name[i]);
        Write('Sales : '); ReadLn(Sales[i]);
    end;{FOR}
    sum := 0.0;
    for i := 1 to n do begin
        sum := 1 sum + Sales[i];
    end; {FOR}
    AvgSales := sum/NoSMen;
    for i := NoSMen do begin
        if (Sales[i] > AvgSales) then begin
            WriteLn(Name[i], ' ', Sales[i]);
        end; {FOR}
    end;{FOR}
    WriteLn('Total Sale for ', NoSMen, ' salesmen = ', sum);
    WriteLn('Average Sales = ', AvgSales);
end. {Prog SalesCalc2}

```

## 8.2.2 Application of Arrays

**1. Histograms or Bar Charts** Given a set of data we wish to count the number of data elements that fall into specified ranges. Here are some examples :

1. A set of ages : how many are between the ages [0–4], [5–9], [10–14], . . . , [95–99]?
2. A set of salaries : how many between [0–1999], [2000–3999], . . . , [98000–99999]?
3. A set of exam marks : how many between [0–9], [10–19], . . . , [90–99]?
4. A set of names 10 characters long : how many begin with 'A', 'B', . . . , 'Z'?

We begin with a naive solution to problem 3—student exam marks. Assume the marks are stored in an array `Mark[1..maxstuds]` and the counts are in `Hist[1..maxints]`.

```

program Histogram1;
const
    MaxStuds = 2000;
    MaxInts  = 50;
type
    MarkType   = 0..100;
    CountType  = 0..MaxStuds;
    MarkArrayType = array[1 . . MaxStud] of MarkType;
    HistoType   = array[1..MaxInts] of CountType;
var
    Mark      : MarkArrayType;
    Hist      : HistoType;
    NoStuds   : 1..MaxStuds;
    i         : 0..MAXINT;
begin
    < Read values into NoStuds and Mark array >
    for i := 1 to MaxInts do begin
        Hist[i] := 0;
    end;
    for i := 1 to NoStuds do begin
        if (Mark[i] >= 0) AND (Mark[i] < 10) do begin
            Hist[1]:= Hist[1] + 1;
        end
        else if (Mark[i] >= 10) AND (Mark[i] < 20) do begin
            Hist[2]:= Hist[2] + 1;
        end
        .
        .
        .
        else begin
            Hist[10]:= Hist[10] + 1;
        end;
    end;{FOR}
    < etc >
end.

```

This is a long program with a lot of **if**-statements that are essentially the same. The purpose of the **if**-statements is to find out which box of the histogram  $\text{Mark}[i]$  falls into. Notice the pattern in the table below :

Mark[i]	Hist box
52	6
09	1
93	10
38	4

The rule is this : if the first digit of  $\text{Mark}[i]$  is  $k$  then it falls into box  $k + 1$ . The first digit is

Mark[i] *div* 10. Incidentally, the second digit is Mark[i] *mod* 10, but this works only for 2-digit numbers.

**Exercise 8.2.1 : Digit Extraction :** How do you extract the *i*th digit from an *n*-digit decimal number?  
□

We can now re-write Histogram1 using this idea to get rid of *all* the **if**-statements.

```

program Histogram2;
  < same as Histogram1 >
BEGIN
  for i := 1 to MaxInts do begin
    Hist[i] := 0;
  end { FOR i }

  for i := 1 to NoStuds do begin
    BoxNo := (Mark[i] div 10) + 1;
    Hist[BoxNo] := Hist[BoxNo] + 1;
  end;{FOR}

  {'Graphical Output'}
  for i := 1 to MaxInts do begin
    WriteLn;
    Write(i-1); Write('|');
    for j := 1 to Hist[i] do begin
      Write('*');
    end;{for j}
    WriteLn(Hist[i]);
  end;{for i}
end.{Prog Histogram2}

```

The output form this program should look like this :

```

0|***3
1|*****5
2|*****9
3|****4
4|****4
5|*****12
6|*****16
7|*****6
8|****4
9|*1

```

The other problems above are solved in the same way, the only difference being in the calculation of the box number. Here are the formulas for the box numbers :

1. A set of ages : how many are between the ages [0–4], [5–9], [10–14], . . . , [95–99]?  
Range : [0–99]. Number of intervals = 20 and interval width = 5.  
BoxNo := (Age[i] div 5) + 1
1. A set of salaries : how many between [0–1999], [2000–3999], . . . , [98000–99999]?  
Range : [0–99999]. Number of intervals = 50. Interval width = 2000.  
BoxNo := (Salary[i] div 2000) + 1
2. A set of exam marks : how many between [0–9], [10–19], . . . , [90–99]? Range :  
[0–99]. Number of intervals = 10. Interval width = 10.  
BoxNo := (Mark[i] div 10) + 1
3. A set of names 5 characters long : how many begin with ‘A’, ‘B’, . . . , ‘Z’?  
Range : [‘AAAAA’–‘ZZZZZ’]. Number of intervals = 26. Interval width = 4<sup>26</sup>.  
BoxNo := Ord(Name[i][1]) - Ord(‘A’) + 1

**Exercise 8.2.1 :** *General Histogram.* Given any set of  $n$  numeric values, write a Pascal program to construct a histogram with  $k$  intervals. □

### 8.2.3 Matrices and Vectors

**Vector and Matrix Representations** Vectors and Matrices are naturally represented by 1- and 2-dimensional arrays in Pascal. We define two new types to represent vectors and matrices as follows :

```

const
    MaxRows = 100;
    MaxCols = 100;
type
    IndexTypeR = 1..MaxRows;
    IndexTypeC = 1..MaxCols;
    RVecType   = ARRAY[IndexTypeR] OF REAL;
    RmatType   = ARRAY[IndexTypeR, IndexTypeC] OF REAL;

```

**Vector and Matrix Operations** We wish to write procedures to implement the following operations on vectors and matrices :

1. 
$$z = ax + y, \text{ where } z_i = ax_i + y_i, i = 1, 2, \dots, n$$

2. 
$$a = x^T y, \text{ where } a = \sum_{i=1}^n x_i y_i, i = 1, 2, \dots, n$$

3. 
$$b = Ax, \text{ where } b_i = \sum_{j=1}^n a_{ij} x_j, i = 1, 2, \dots, n$$

4.

$$b = xA, \text{ where } b_j = \sum_{i=1}^n a_{ij}x_i, \quad i = 1, 2, \dots, n$$

5.

$$C = A + B, \text{ where } c_{ij} = a_{ik}b_{kj}, \quad i, j = 1, 2, \dots, n.$$

6.

$$C = A \times B, \text{ where } c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}, \quad i, j = 1, 2, \dots, n.$$

```

PROCEDURE SVecVecAdd(VAR x,y,z :RvecType; a:REAL;
                    n :IndexRType);

var
  i : IndexRType;
begin
  FOR i := 1 TO n DO BEGIN
    z[i] := a*x[i] + y[i];
  END; { FOR i }
END;{ PROC SVecVecAdd }

```

```

FUNCTION VecVecMult(VAR x,y :RvecType) : REAL;
                    n :IndexRType);

VAR
  i : IndexRType;
BEGIN
  sum := 0.0;
  for i := 1 TO n DO BEGIN
    sum := sum + x[i] * y[i];
  end; { FOR i }
  VecVecMult := sum;
end;{ FUNC VecVecMult }

```

```
PROCEDURE MatVecMult(VAR A :RMatType;
                    VAR x,b :RvecType;
                    n :IndexRType);

VAR
  i,j : IndexRType;
  sum : REAL;
BEGIN
  FOR i := 1 TO n DO BEGIN
    sum := 0.0
    FOR j := 1 TO n DO BEGIN
      sum := sum + A[i,j]*x[j];
    END; { FOR j }
    b[i];
  END; { FOR i }
END;{ PROC MatVecMult }
```

```
PROCEDURE VecMatMult(VAR A :RMatType;
                    VAR x,b :RvecType;
                    n :IndexRType);

VAR
  i,j : IndexRType;
  sum : REAL;
BEGIN
  FOR j := 1 TO n DO BEGIN
    sum := 0.0
    FOR i := 1 TO n DO BEGIN
      sum := sum + x[i]*A[i,j];
    END; { FOR i }
    b[j];
  END; { FOR j }
END;{ PROC VecMatMult }
```

```
PROCEDURE MatrixAdd(VAR A,B,C :RMatType;
                    n :IndexRType);

VAR
  i,j : IndexRType;
BEGIN
  FOR i := 1 TO n DO BEGIN
    FOR j := 1 TO n DO BEGIN
      C[i,j] := A[i,j]+B[i,j];
    END; { FOR j }
  END; { FOR i }
END;{ PROC MatrixAdd }
```

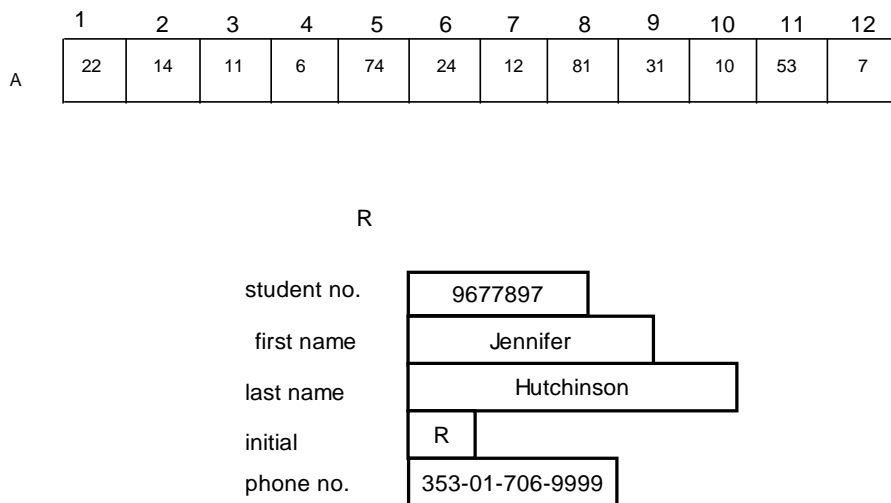
```

PROCEDURE MatrixMult(VAR A,B,C :RMatType;
                    n :IndexRType);
VAR
    i,j,k : IndexRType;
    sum   : REAL;
BEGIN
    FOR i := 1 TO n DO BEGIN
        FOR j := 1 TO n DO BEGIN
            sum := 0.0;
            FOR k := 1 TO n DO BEGIN
                sum := sum + A[i,k]*B[k,j];
            END; { FOR k }
            C[i,j] := sum;
        END; { FOR j }
    END; { FOR i }
END;{ PROC MatrixMult}

```

### 8.2.4 Records

An array is a structured type whose components are all the same type. A record is a structured type whose components may be different types. Figure 7.4 show the difference between the two structured types.



**Figure 8.3.** An Array and a Record.

An individual component of the array **A** above is identified as **A[i]** whereas an individual component of a record **R** is identified as , for example, **R.LName** which is the component of the record that holds the last name.

### Syntax of the RECORD Type

```
type
  <Record type-identifier> = record
      <fieldname 1> : <type 1>;
      <fieldname 2> : <type 2>;
      .
      .
      .
      <fieldname k> : <type k>;
  end
```

where <type 1> etc. are any previously-defined types.

EXAMPLE : *Record Definitions.*

```
type
  StudRecType = record
      IdNo : 1..maxint;
      FName : STRING[10];
      LName : STRING[15];
      Init : CHAR
      Phone : STRING[20];
  end;

  DateType = record
      d : 1..31;
      m : 1..12;
      y : 1900..20000;
  end;

var
  Student : StudRecType;
  DofBirth : DateType
```

□

**Using Records** The following partial program shows how to use record and access individual components

EXAMPLE : *Using Records* .

```
VAR
  Student1 : StudRecType;
  Student2 : StudRecType;
  Student3 : StudRecType;
  DofBirthX,
  DofBirthY : DateType;

BEGIN
  Student1.FName := 'John';
  Student1.LName := 'JONES';
  Student1.Idno := 1234567;

  Student2.FName := 'Tom';
  Student2.LName := Student1.LName;

  DofBirthX.d := 18;
  DofBirthX.m := 11;
  DofBirthX.y := 1971;

  DofBirthY := DofBirthX; { whole-record assignment}

  Readln(Student3.FName);
  Readln(Student3.LName);

  Write(Student1.Idno);
```

□

## 8.2.5 Combined Data Structures

Pascal's powerful facility for defining new data types allows the programmer to combine data structures to form new data structures of any level of complexity. For example, we may have records with components that are arrays; arrays whose components are records; records whose components are records; arrays whose components are arrays, and so on. Care must be taken however : new data structures should not be more complex than necessary. For example, never use a two-dimensional array when 2 one-dimensional arrays will do.

We now show by way of examples how to combine and use data structures.

EXAMPLE : *Records with Array components.*

```
TYPE
  StudRecType = RECORD
    IdNo      : 1..maxint;
    Name      : STRING;
    Mark      : ARRAY[1..8] OF REAL;
    AvgMark   : REAL
  END;

VAR
  Student : StudRecType;
  i       : 1..100;
  sum     : REAL;

BEGIN
  Readln(Student.Name);
  Readln(Student.Idno);

  FOR i := 1 TO 8 DO BEGIN
    Readln(Student.Mark[i]);
  END;

  sum := 0.0;
  FOR i := 1 TO 8 DO BEGIN
    sum := sum + Student.Mark[i];
  END;

  Student.AvgMark := sum/8.0;
```

□

Notice that `Student.Mark` is an array of 8 reals and `Student.Mark[i]` is the  $i$ th component of this array.

EXAMPLE : *Arrays of Records.* These are useful when we want to use a multi-column table.

```

PROGRAM OverDrawnList;
{ This program writes out a list of overdrawn customers}
CONST
    Maxcusts = 1000;
TYPE
    CustRecType = RECORD
        AccNo    : 1..maxint;
        Name     : STRING;
        Addr     : STRING;
        Balance  : REAL
    END;
    CustTableType = ARRAY[1..maxcusts] OF CustRecType;
VAR
    CustTable : CustTableType;
    i, NoCusts : 1..maxcusts;
BEGIN
    < Read Data into CustTable >
    FOR i := 1 TO NoCusts DO BEGIN
        IF(CustTable[i].Balance < 0.0) THEN BEGIN
            Write(CustTable[i].AccNo);
            Write(CustTable[i].Name);
            Write(CustTable[i].Balance);
            Writeln;
        END; {FOR}
    
```

□

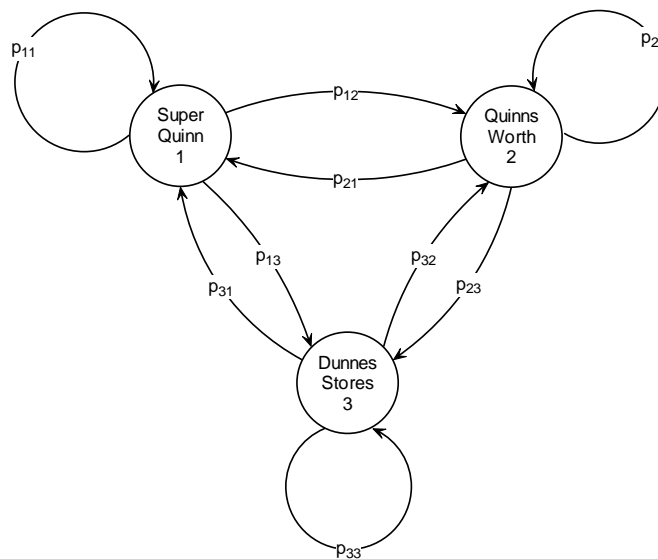
AccNo	Name	Balance	Address

Figure 8.4. Customer Record Table.

**Laboratory Exercise No. 8: Market Share Analysis & Markov Chains.**

**Background.** We wish to analyse the market share and customer loyalty for three supermarkets : SuperQuinn, Quinnsworth, and Dunne’s Stores. To simplify things, let us assume that any customer shops once a week in one, and only one of the three stores. Any customer can be in one of three states at any time :

1. Last shopped at Superquinn (State 1)
2. Last shopped at Quinnsworth (State 2)
3. Last shopped at Dunne’s Stores (State 3)



**Figure 8.5.** Customer Switching.

Customers are not completely loyal to the supermarket they currently shop at and they may switch from one store to another each week. Let us define the following probabilities :

$p_i^k$  = the proportion of all customers that shop in store  $i$  in week  $k$ .

$p^k = [p_1^k \ p_2^k \ p_3^k]$  the *state-proportion vector* in week  $k$ . This gives the *market share* for each store in week  $k$ .

$p_{ij}$  = the probability that a customer who shopped in store  $i$  this week switches to store  $j$  next week.

$P$  = the *transition probability matrix*

$$P = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}$$

that gives the probability of a customer switching from any store to any other store. Note for each row  $i$  the transition matrix  $P$  must satisfy the condition  $\sum_{j=1}^n p_{ij} = 1$ , i.e., the probabilities in any row must sum to 1. Why?

The market share vector  $p^k$  for week  $k$  may be calculated from the previous week's vector  $p^{k-1}$  as

$$p^k = p^{k-1}P, \quad \text{or} \quad [p_1^k \quad p_2^k \quad p_3^k] = [p_1^{k-1} \quad p_2^{k-1} \quad p_3^{k-1}] \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}$$

We want to find out the market share of each store in the long run. If we start with some initial market share vector  $p^1 = [1/3 \quad 1/3 \quad 1/3]$ , say, then if we calculate the vector-matrix product  $p^k = p^{k-1}P$  for  $k = 2, 3, \dots$  then we get a sequence of market share vectors  $p_1, p_2, \dots, p_k$  that eventually converges to a constant vector  $p = [p_1 \quad p_2 \quad p_3]$ , and this is the long-run market share vector.

The answer to this problem is a solution of the following matrix equation

$$[p_1 \quad p_2 \quad p_3] = [p_1 \quad p_2 \quad p_3] \begin{bmatrix} 0.8 & 0.05 & 0.15 \\ 0.10 & 0.75 & 0.15 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

One way to solve this problem is to generate a sequence of  $p$  vectors as follows :

$$[p_1^{k+1} \quad p_2^{k+1} \quad p_3^{k+1}] = [p_1^k \quad p_2^k \quad p_3^k] \begin{bmatrix} 0.8 & 0.05 & 0.15 \\ 0.10 & 0.75 & 0.15 \\ 0.1 & 0.1 & 0.8 \end{bmatrix},$$

starting with

$$[p_1^0 \quad p_2^0 \quad p_3^0] = [1/3 \quad 1/3 \quad 1/3]$$

The first few iterations gives

$$\begin{aligned} [p_1^0 \quad p_2^0 \quad p_3^0] &= [1/3 \quad 1/3 \quad 1/3] \\ [p_1^1 \quad p_2^1 \quad p_3^1] &= [0.3333 \quad 0.3000 \quad 0.3667] \\ [p_1^2 \quad p_2^2 \quad p_3^2] &= [0.3333 \quad 0.2783 \quad 0.3883] \\ [p_1^3 \quad p_2^3 \quad p_3^3] &= [0.3333 \quad 0.2642 \quad 0.4024] \end{aligned}$$

The  $p$ -vector sequence converges to

$$[p_1 \quad p_2 \quad p_3] = [0.3333 \quad 0.2381 \quad 0.4286]$$

The interpretation of this result is : in the long run Superquinn has 33.33%, Quinnsworth has 23.81% and Dunne's Stores has 42.86% of the market, for people who shop at these stores. Notice the curious result : SuperQuinn's market share does not change over time.

**Laboratory Exercise.** Program all the matrix-vector procedures discussed in class and in the notes and test them to ensure that they run properly.

Use these matrix-vector procedures to solve the market share problem above. That is, write a program with the following specification :

1. **Input** : Prompt the user to type in (i) the number of stores  $n$  (ii) the initial market-share vector  $p^0$ , and (iii) the state transition matrix  $P$ .
2. **Processing** : Perform the following step until the long-run market share vector has been found : call the *vector-matrix multiplication* procedure, sending down to it the vector  $p_{old}$ , the matrix  $P$ , and getting back the vector  $p_{new}$ . At the next iteration the new vector becomes the old, and the process repeats.
3. **Output** : Write out the new vector at each iteration along with the iteration number. At the end of the program write out the final market share vector along with an explanation of the numbers.

This laboratory exercise will cover the next two Fridays, 12 and 19 Nov 1999. There will be no laboratory class in the last week of this semester, Nov 26.

---

---