Lecture 9

Suppose we have an array int f [10]
which already has values in it.

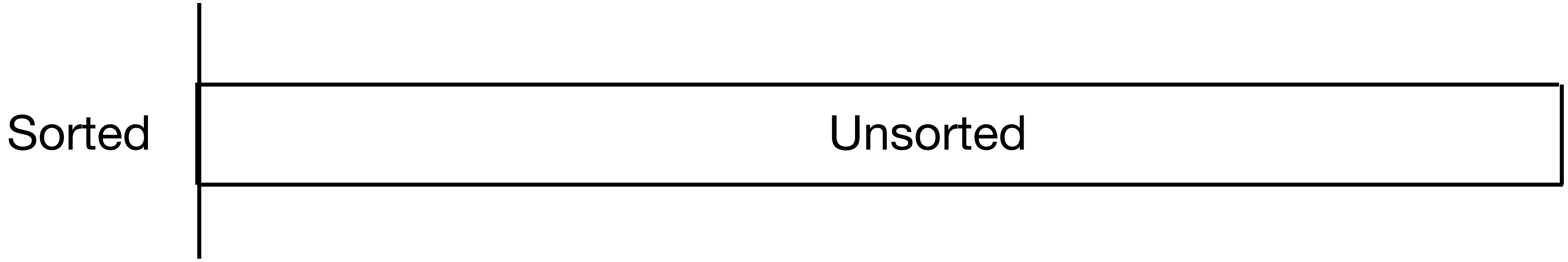
We would like to rearrange those values so that they were ascending.

This is called sorting the array into ascending order.

There are many different algorithms that can be used for sorting.

Today we look at 2 of them.

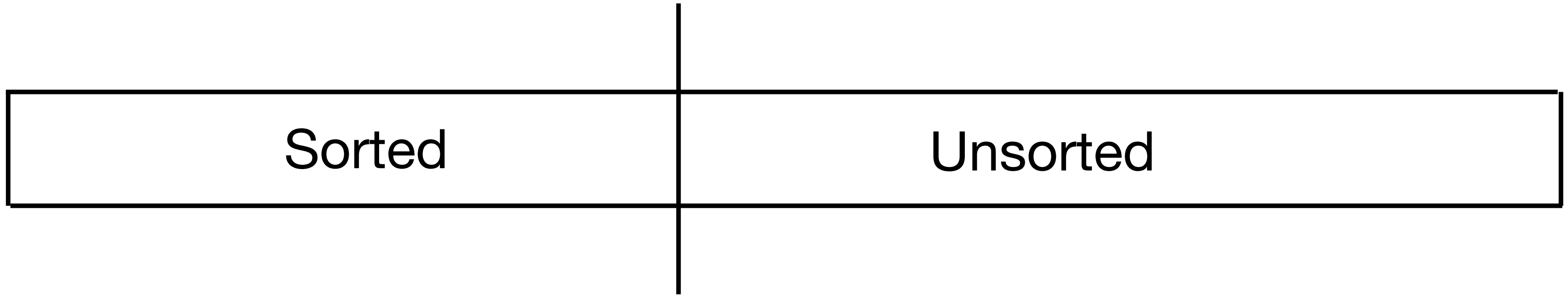
Before we start (Sorted part is empty)



Sorted

Unsorted

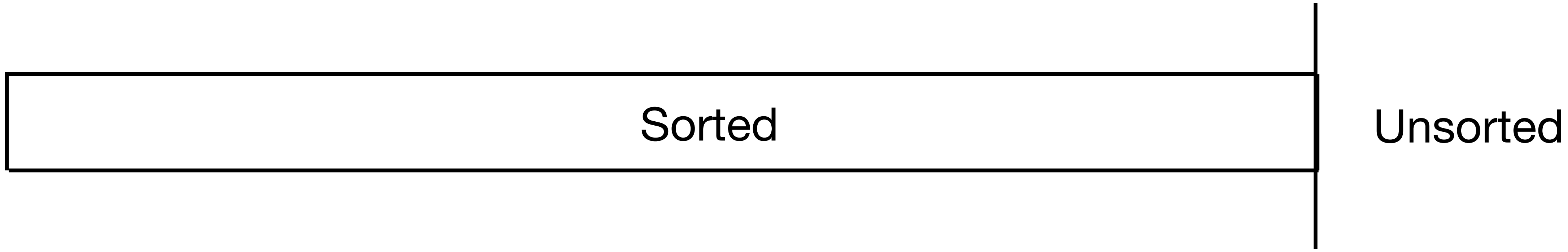
During the sorting (Some sorted and some unsorted)



Sorted

Unsorted

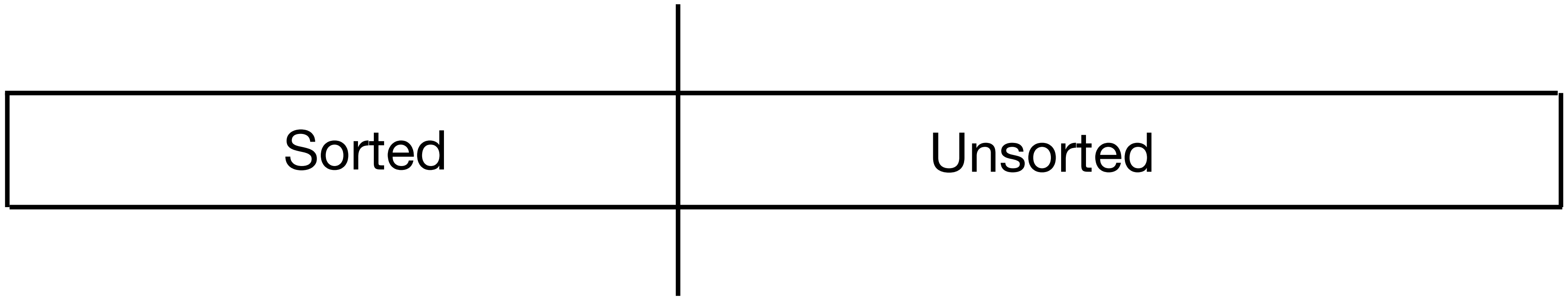
When we finish (Unsorted part is empty)



Sorted

Unsorted

To make progress we need to move the line to the right.



Sorted

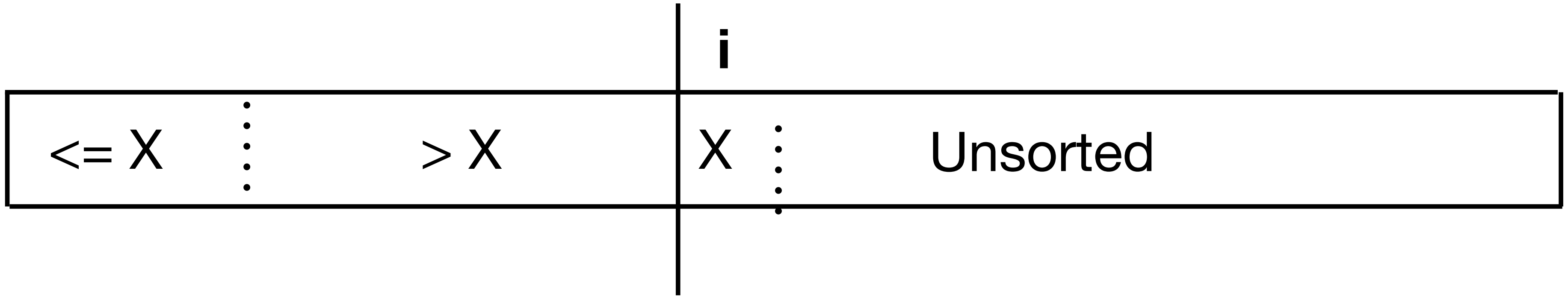
Unsorted

How should we do this?

Everything to the left of the line is sorted and we are currently looking at f[i]. Let is call the value in there X.

In the Sorted part of the array we can imagine it is divided into 2 parts those sorted values which are <= X and those sorted values that are > X.

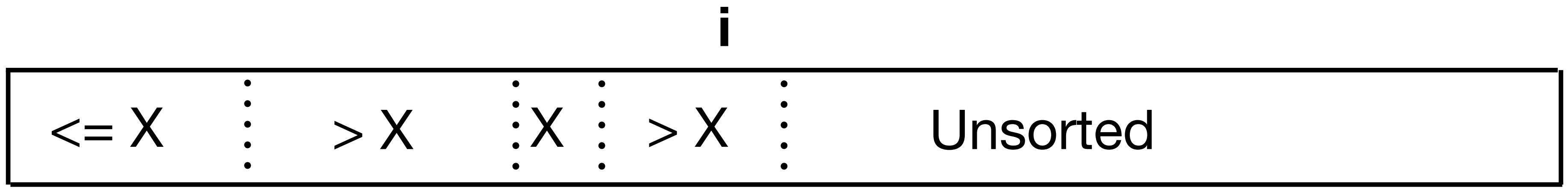
Remember that either of those could be empty.



We would like to insert the value at f[i] into the correct place so that everything up to and including f[i] is now sorted.

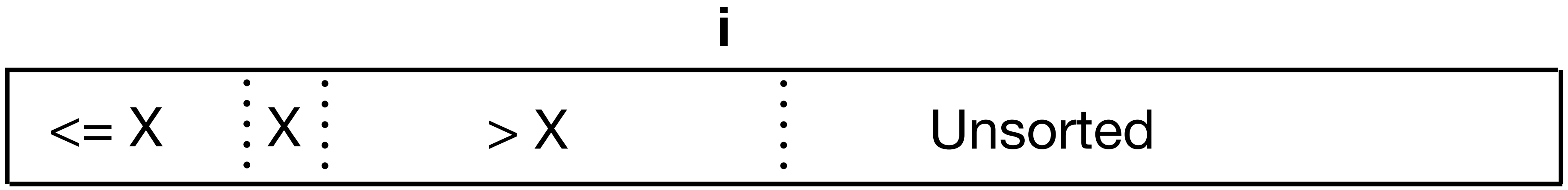
```
i = 0 ;
while (i != 10)
{
   “Insert f[i] into correct place so
    that f[0..i] is sorted”
   i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

During the process we might have a picture like this



It seems that the value X is moving down through the group of values that are greater than it.

We will finish when X finally reaches the start of the section which are <=X



We use a function which takes the addresses of 2 ints and swaps the values at those addresses.

```
void swap (int *x, int *y)
{
        int temp ;
        temp = *x ;
        *x = *y ;
        *y = temp;
}
```

```
int i ;
int j ;

i = 0 ;
while (i != 10)
{
   j = i ;
   while (( j != 0 ) && ( f[j] < f[j-1] ))
   {
      swap( &f[j], &f[j-1] ) ;
      j = j-1;
   }
   i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

Note.

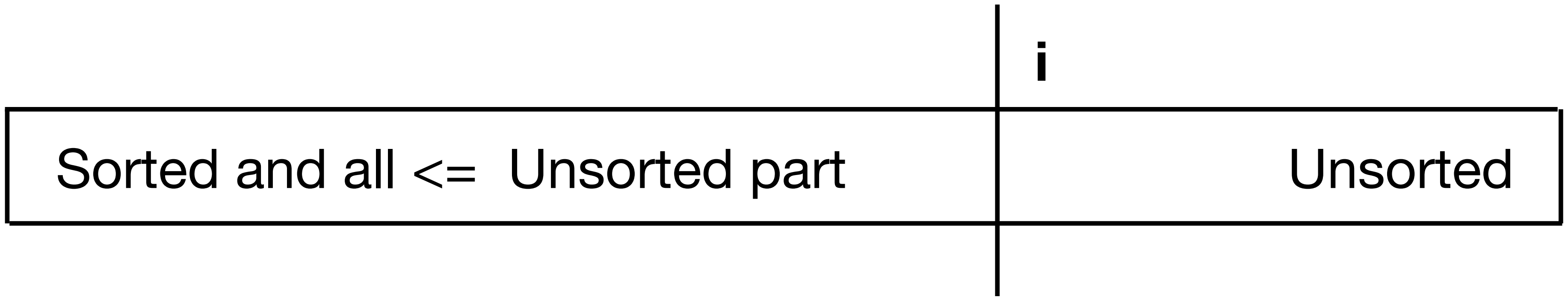
```
   while (( j != 0 ) && ( f[j] < f[j-1] ))
```

When j == 0 we hope that we never get to
test f[j] < f[j-1] because that would be out of bounds.

In C if we have C1 && C2
Then C2 is only tested if C1 is true.

Be careful as not all programming languages evaluate boolean expressions
in this way.

Let us go back a few slides. We were looking at how to extend the sorted section of the array so as to make progress and move the line to the right.



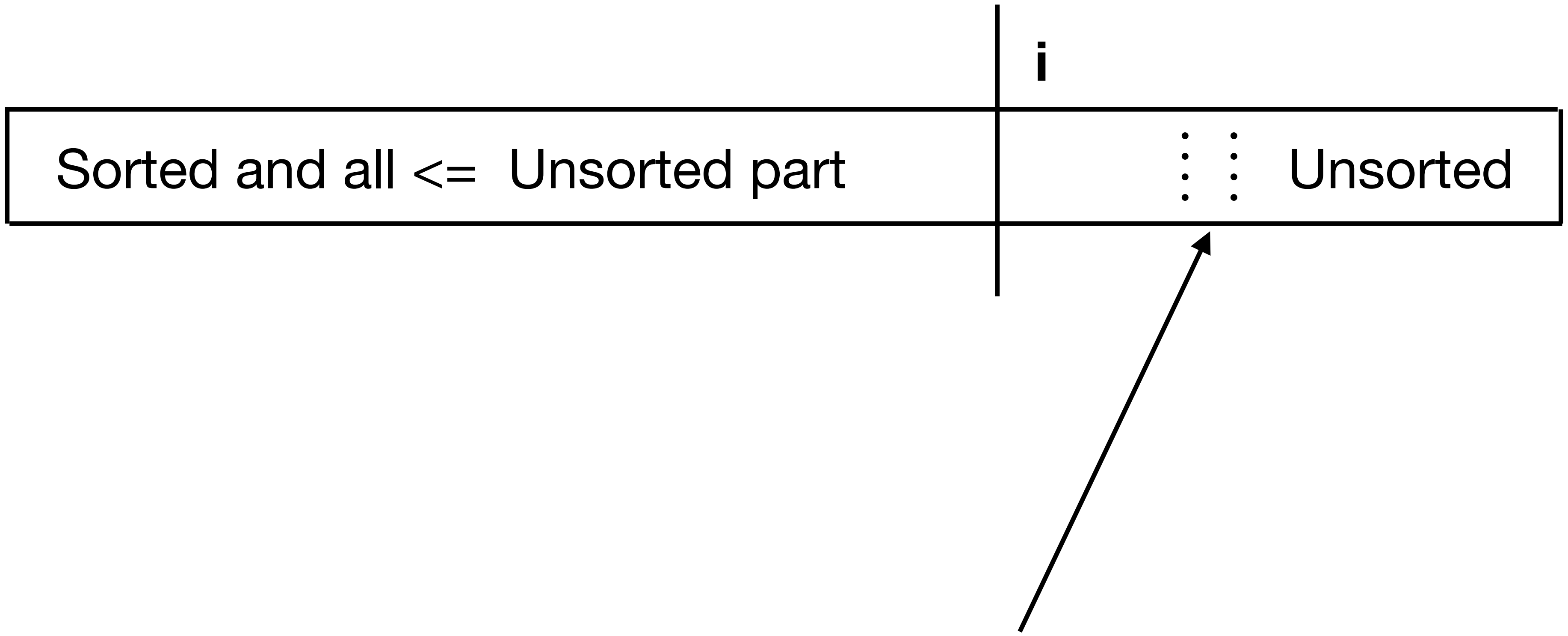
i

Sorted and all <= Unsorted part

Unsorted

Let us change our picture a little and make the part to the left of the line all sorted but also all <= to the values in the Unsorted part

Now to make progress we need to find the smallest value in the Unsorted part of the array and swap it with the value in f[i]. Then we can extend the Sorted part to include f[i]



Smallest value in the Unsorted part

```
i = 0 ;
while (i != 10)
{
  “Find the smallest values in the unsorted part
  and swap it with the value in f[i]”
  i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

```
int i ;
int j ;
int min_index ;
i = 0 ;
while (i != 10)
{
 j = i ;
 min_index = i ;
 while ( j!= 10)
 {
     if (f[j] <= f[min_index])
         { min_index = j ; }
     j = j+1 ;
 }
// f[min_index] is smallest value in unsorted part

 swap(&f[i], &f[min_index]) ;
 i = i+1 ;

}

// everything up to position i is sorted and i == 10
```

Insertion Sort

```
int i ;
int j ;
int min_index ;
i = 0 ;
while (i != 10)
{
 j = i ;
 min_index = i ;
 while ( j!= 10)
 {
     if (f[j] <= f[min_index])
         { min_index = j ; }
     j = j+1 ;
 }
 swap(&f[i], &f[min_index]) ;
 i = i+1 ;
}
// everything up to position i is sorted and i == 10
```

Selection sort

```
int i ;
int j ;

i = 0 ;
while (i != 10)
{
  j = i ;
  while (( j != 0 ) && ( f[j] < f[j-1] ))
  {
    swap( &f[j], &f[j-1] ) ;
    j = j-1;
  }
  i = i+1 ;
}

// everything up to position i is sorted and i == 10
```

Sorting and speed

Suppose your array contains N elements

If you use either of the Sorting programs we have looked at then the outer loop will be performed N times.

For each of these, the inner loop will be performed on average N/2 times.

So the time it takes to run the program will be proportional to N * (N/2)

We usually write this as O(N*N)

This is called the Time Complexity or the Temporal Complexity of the program

As N gets bigger N*N gets bigger faster.

Sorting and speed

N	N^2	N^3	
10	100	1000	
100	10000	1000000	
1000	10^6	10^9	
10000	10^8	10^12	

If a computer can perform 1,000,000 instructions per second then a program which has 10^12 instructions would need 1,000,000 seconds to run, that is 16,666 minutes, or 277 hours, or about 5 and a half days.

Complexity and speed

The best ways to improve the speed of a program are to try to redesign it so as to go from O(N^3) to O(N^2) or from O(N^2) to O(N).

In future modules we will study some techniques to allow you to do this.

Saving a few iterations in a loop really doesn’t make much of a difference, so don’t try too hard to improve efficiency that way.

Multi-dimensional arrays

- We have seen how to declare a 1-dimensional array in C

- int f[10] ; this declares an array called f which has 10 elements numbered from 0 to 9

- The elements are f[0], f[1], ..., f[9]

Multi-dimensional arrays

- Arrays can have more than 1 dimension.
- int f [3] [4] ; this declares a 2-dimensional array of integers
- Let is write some statements to declare an array like this and then read values into it.

```
void main ()
{
    int f [3] [4] ;

    int i ;
    int j ;

    i = 0;
    while ( i != 3)
    {

//      read values into row i

        i = i+1 ;
    }
}
```

```
void main ()
{
    int f [3] [4] ;

    int i ;
    int j ;

    i = 0;
    while ( i != 3)
    {
        j = 0 ;
        while ( j != 4)
        {
            scanf( “%d”, &f[i][j] ) ;
            j = j+1 ;
        }
        i = i+1 ;
    }
}
```

Matrix addition

We can use 2-dimensional arrays to represent matrices.

Let us write some statements to add two 3 X 4 matrices.

```
void main ()
{
        int a [3] [4] ;
        int b [3] [4] ;
        int c [3] [4] ;

        int i ;
        int j ;

// statements to read values into a and b.

        i = 0;
        while ( i != 3 )
        {
                j = 0 ;
                while ( j != 4 )
                {
                        c[i] [j] = a[i] [j] + b[i] [j] ;
                        j = j+1 ;
                }
                i = i+1 ;
        }
}
```

Matrix multiplication

If we have 2 matrices a [3] [4] and b [4] [2] then we can multiply them
because the 2nd dimension of matrix a is equal to the first dimension of matrix b.

Let us write some statements to multiply these two matrices.

```
void main ()
{
    int a [3] [4] ;
    int b [4] [2] ;
    int c [3] [2] ;

    int i ;
    int j ;

// statements to read values into a and b.

    i = 0;
    while ( i != 3 )
    {
        j = 0 ;
        while ( j != 2 )
        {

//          c[i] [j] = ”the values in row i of matrix a multiplied by the
//                  values in column j in matrix b “

            j = j+1 ;
        }
        i = i+1 ;
    }
}
```

```
void main ()
{
        int a [3] [4] ;
        int b [4] [2] ;
        int c [3] [2] ;

        int i ;
        int j ;
        int k ;

// statements to read values into a and b.

        i = 0;
        while ( i != 3 )
        {
                j = 0 ;
                while ( j != 2 )
                {
                        k = 0 ;
                        c [i] [j] = 0 ;
                        while ( k != 4 )
                        {
                                c [i] [j] = c [i] [j] + ( a[i] [k] * b [k] [j] ) ;
                                k = k+1 ;
                        }
                        j = j+1 ;
                }
                i = i+1 ;
        }
}
```

Matrix display

If we have a 3 X 4 matrix we might want to display print the values in a nicely formatted way.

```
void main ()
{
    int a [3] [4] ;

    int i ;
    int j ;

// statements to read values into a.

    i = 0;
    while ( i != 3 )
    {
        j = 0 ;
        while ( j != 4 )
        {
            printf( “%d”, a [i] [j] ) ;
            j = j+1 ;
        }
        printf( “\n” ) ;
        i = i+1 ;
    }
}
```

Problem

- Construct a program which will read in 100 values each of which is in the range from 0 to 19, and will count the number of times each of the values occurs.

Problem

- We have to count how many times each of the 20 values occur.
- We could declare 20 different variables to use as counters
- Or, we could use an array with 20 elements. We will choose to use this approach.

```
void main ()

        int count [20] ;
        int i ;
        int value ;

        i = 0 ;
        while ( i != 20 )
        {
                count [ i ] = 0 ;
                i = i +1 ;
        }

        i = 0 ;
        while ( i != 100 )
        {
                scanf( “%d”, &value ) ;

//              count this value

                i = i+1;
        }
```

```
void main ()

    int count [20] ;
    int i ;
    int value ;

    i = 0 ;
    while ( i != 20 )
    {
        count [ i ] = 0 ;
        i = i +1 ;
    }

    i = 0 ;
    while ( i != 100 )
    {
        scanf( “%d”, &value ) ;

        count [value] = count [value]+1 ;

        i = i+1;
    }
```

Problem

- Suppose we want to do some statistics concerning the height of people in a village.
- We are told that there are 3000 people in the village.
- We want to record their heights in the following ranges (all values in cm).
- 0 - 39, 40 - 79, 80 - 119, 120 - 159, 160 - 199, 200 - 239

Problem

- We need to count the number of people in each of 6 categories.

- We take a similar approach to the last problem and use an array to store our counts.

```
void main ()

        int count [6] ;
        int i ;
        int value ;

        i = 0 ;
        while ( i != 6 )
        {
                count [ i ] = 0 ;
                i = i +1 ;
        }

        i = 0 ;
        while ( i != 3000 )
        {
                scanf( “%d”, &value ) ;

//              count this value

                i = i+1;
        }
```

```
void main ()

    int count [6] ;
    int i ;
    int value ;

    i = 0 ;
    while ( i != 6 )
    {
        count [ i ] = 0 ;
        i = i +1 ;
    }

    i = 0 ;
    while ( i != 3000 )
    {
        scanf( “%d”, &value ) ;

        count[ value/40 ] = count [ value/40 ] + 1 ;

        i = i+1;
    }
```

Problem

- A “magic square” has the property that adding any row, or any column, or either diagonal given the same total
- Magic squares were discovered in China around 650 bc.
- I want to play with “odd” magic squares, they are n X n where n is an odd value.

8	**1**	**6**
3	**5**	**7**
4	**9**	**2**

each row and each
column and the
main diagonals all
add up to 15.

Problem

- Can you find a method to construct a 3 X 3 magic square?
- Given a magic square how many others could you make from it?