Appendix A

In this appendix, we will cover some additional recipes related to sorting, which we covered in `Chapter 9`, *Sorting and Searching*:

- Arranging numbers in ascending order using selection sort
- Arranging numbers in ascending order using merge sort
- Arranging numbers in ascending order using shell sort
- Arranging numbers in ascending order using radix sort

Arranging numbers in ascending order using selection sort

In this sorting method, the first value is picked up and compared to all of the remaining values in the array. If a value smaller than the first value is found, their places are interchanged. Consequently, in the first iteration, the smallest value in the array is accessed and assigned to the first location of the array. This process is repeated with the second value in the array. As a result, the second smallest value will be placed at the second location in the array. This procedure is repeated until we have the values in ascending order.

How to do it…

Let's assume that the function to do selection sort is `SelectionSort`, which can be invoked using the following parameters:

```
SelectionSort (arr,n)
```

Here, `arr` is the array comprising of `n` elements to be sorted:

1. Store the ith value of the array in a variable, `min`.
2. Also, store the location of the `min` value in the variable called `loc`, that is, `loc=i`.

3. Compare the value in the `min` variable with the rest of the array elements, beginning from `i+1`. So, repeat *step 5* for `j=i+1` to *n* times, that is, for `j=i+1, i+2,....n`.
4. Find out the minimum value, as well its location, in the array from the i+1th position until the end of the array.
5. Interchange the values of `arr[i]` with `arr[loc]`.
6. Repeat *steps 1* to *5* for `n-1` times; that is, for `i=1,2,....n-1`.

The program for sorting the elements of an integer array using the selection sort technique is as follows:

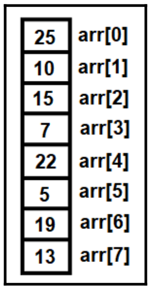
```
//selectionsort.c

#include<stdio.h>

#define max 20

int main() {
  int arr[max], i, j, value, len, index, temp;
  printf("How many numbers to sort? ");
  scanf("%d", & len);
  printf("Enter %d values:\n", len);
  for (i = 0; i < len; i++)
    scanf("%d", & arr[i]);
  for (i = 0; i < len - 1; i++) {
    value = arr[i];
    index = i;
    for (j = i + 1; j < len; j++) {
      if (value > arr[j]) {
        value = arr[j];
        index = j;
      }
    }
    temp = arr[i];
    arr[i] = arr[index];
    arr[index] = temp;
  }
  printf("\nThe ascending order of the element entered is:\n");
  for (i = 0; i < len; i++)
    printf("%d\n", arr[i]);
  return 0;
}
```

Now, let's go behind the scenes to understand the code better.

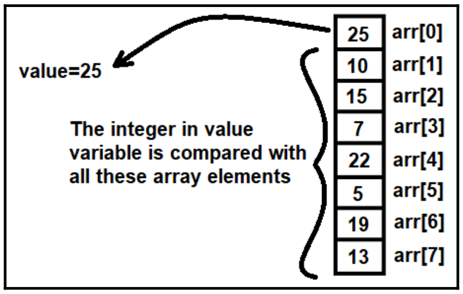
How it works...

Let's define a macro called `max` that has a value of `20` and an integer array, `arr`, that's `max` in size, which will enable the array to accommodate up to 20 integer values. If you think the user may enter more values, you can always increase the value of `max` to any desired value.

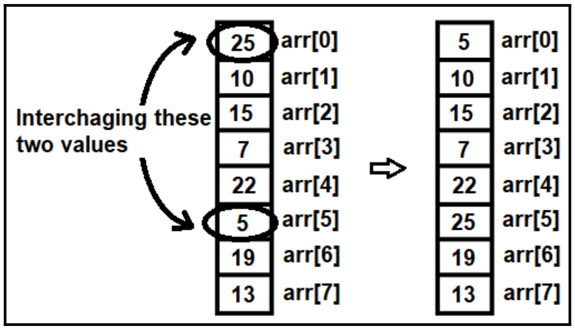
The user will be prompted to enter the numbers that need to be sorted. Let's assume that we want to sort eight values; so, the value `8` will be assigned to the `len` variable. Thereafter, the user will be asked to enter the values that need to be sorted. Let's say the values we entered that are assigned to the `arr` array are as follows:



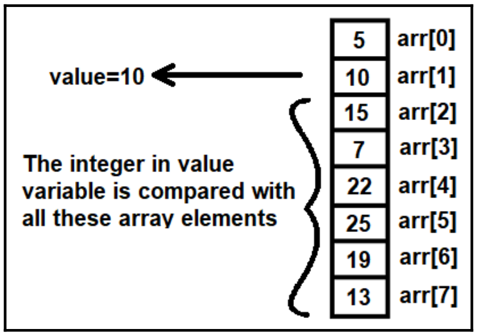
Here, we will employ a nested loop where the outer loop is run via the `i` variable and the inner `for` loop is run via the `j` variable. The outer `for` loop is set to execute from 0 until `len-1` times. In the first iteration, the value of `i` is `0` and the value of `arr[0]`, that is, the first value in the array, is assigned to the variable value. The inner `for` loop is set to execute from `i+1` until < `len` times; that is, the value of `j` will run from 1 until 7. All of the values in the `arr` array from `i+1` to 7 will be compared with the integer in the `value` variable. If any value in the array is smaller than the `value` variable, their locations will be interchanged; that is, the smaller value will be assigned to the `value` variable and the index location of that element will be stored; that is, it will be assigned to the `index` variable:



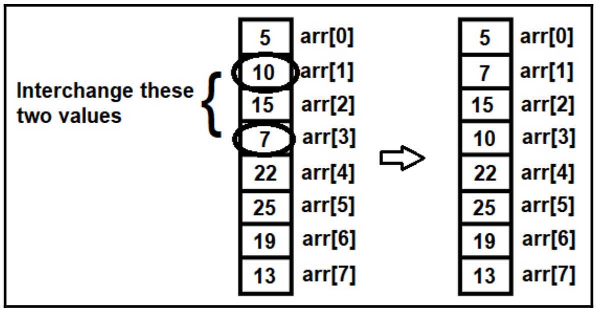
Once we find the smallest value in the array, that value's index location is interchanged with the value at the index location, `arr[0]`, as shown here:



This process is repeated with the second element in the array, that is, the `arr[1]` value is assigned to the `value` variable. Now, the integer in the `value` variable is compared with the integers in the array from index locations `arr[2]` until `arr[7]`. This happens because, after the first iteration, we have the smallest value of the array at its correct location, that is, at `arr[0]`:



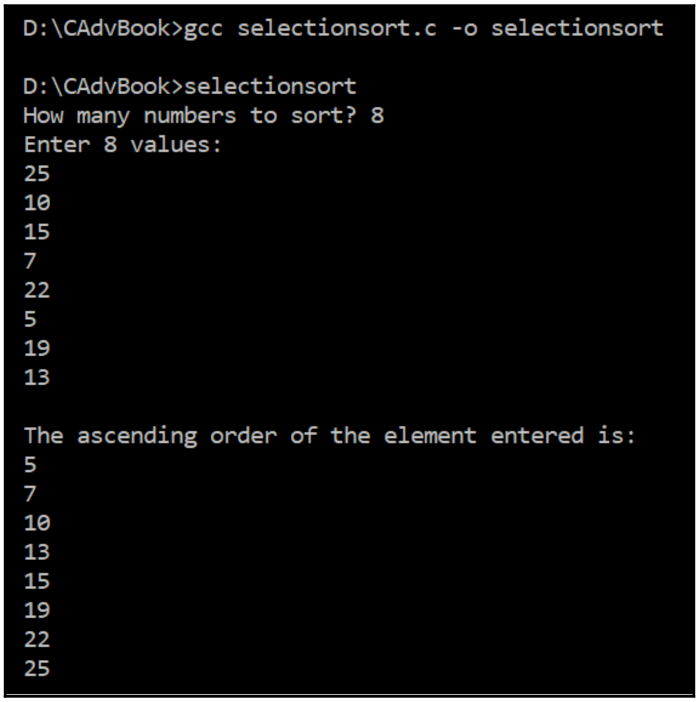
Again, if a smaller integer than the current variable value is found in the array, that integer is interchanged with the `value` variable and the index location of that integer is also assigned to the `index` variable. After executing the inner for `j` loop, the `value` variable will have the smallest integer of the array from index locations `arr[2]` until `arr[7]`, that is, the `value` variable will have the value `7`. Also, the index location of that smallest integer, that is, 3, will be assigned to the `index` variable. Finally, the integer at `arr[1]` will be interchanged with the `arr` element at the index location specified by the `index` variable; that is, integer values `10` and `7` will be interchanged, as shown in the following diagram:



The program is compiled using GCC using the following statement:

```
gcc selectionsort.c -o selectionsort
```

Because no error appears on the compilation, that means the `selectionsort.c` program has successfully compiled into the `selectionsort.exe` file. On execution, it will ask the user to specify how many numbers are to be sorted. Following this, the program will prompt the user to enter the numbers to be sorted. After entering these numbers, they will appear sorted in ascending order, as shown in the following screenshot:



```
D:\CAdvBook>gcc selectionsort.c -o selectionsort

D:\CAdvBook>selectionsort
How many numbers to sort? 8
Enter 8 values:
25
10
15
7
22
5
19
13

The ascending order of the element entered is:
5
7
10
13
15
19
22
25
```

Voilà! We've successfully used selection sort to arrange numbers in ascending order. Now, let's move on to the next recipe!

Arranging numbers in ascending order using merge sort

In this recipe, we will learn how to arrange some integers in ascending order using the merge sort technique. Merge sort is a divide and conquer algorithm. It recursively breaks the array into subarrays until the subarray becomes so small that it consists of a single element. Thereafter, the subarrays are merged or combined into a sorted array.

How to do it...

Let's assume that the merge sorting is done by invoking the following `mergesort` method:

```
mergesort (array arr, start, end)
```

Here, `arr` is the array of `n` elements, the start is the initial range of the array, and the end is the final limit of the array, `arr`.

The following steps will be performed to do a merge sort:

1. If `end > start`, then go to *step 2*; otherwise, exit.
2. Find out what the middle value of the array is.
3. Individually perform `mergesort` (recursively) on the two halves of the array by invoking the following methods:

   ```
   mergesort(arr, start, mid)
   mergesort(arr,mid+1, end)
   ```

4. After the two halves of the array have been sorted, the two sorted halves can be merged by calling the `merge` function with the following parameters:

   ```
   merge(arr,start,mid,mid+1,end)
   ```

The `merge` method, `merge(array arr, lower1, upper1, lower2, upper2)`, will perform the following steps to merge the two sorted halves of the array. These steps are a continuation of the preceding steps, but because the `merge` function is recursively called, the steps of the `merge` method are renumbered from 1:

1. Repeat *steps 2* and *3* while `lower1<upper1` and `lower2 < upper2`.
2. If `arr[lower1] < arr[lower2]`, then insert the element at `arr[lower1]` into another array, say `mgd`; otherwise, go to *step 3*.
3. Insert the element at `arr[lower2]` into the `mgd` array.
4. If the first half of the array is not yet over, the remaining portion is stored in the `mgd` array.
5. If the second half of the array is not yet over, the remaining portion is stored in the `mgd` array.
6. The `mgd` array now contains the merged and sorted version of the two halves of the array. Copy the content of the `mgd` array back into the original array, `arr`.

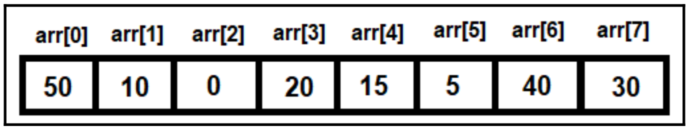
The program for sorting elements of an integer array using the merge sort technique is as follows:

```
//mergesort.c

#include <stdio.h>

#define max 20
void merge(int arr[], int fsi, int fei, int ssi, int sei);
void mergesort(int arr[], int si, int ei);
int main() {
  int arr[max];
  int si, i, len;
  si = 0;
  printf("How many numerical to sort? ");
  scanf("%d", & len);
  printf("Enter %d numerical:\n", len);
  for (i = 0; i <= len - 1; i++)
    scanf("%d", & arr[i]);
  printf("Original array is \n");
  for (i = 0; i < len; i++)
    printf("%d\t", arr[i]);
  printf("\n");
  mergesort(arr, si, len - 1);
  printf("Sorted array is \n");
  for (i = 0; i < len; i++)
    printf("%d\t", arr[i]);
  return 0;
}
void mergesort(int arr[], int si, int ei) {
  int mid;
  if (ei > si) {
    mid = (si + ei) / 2;
    mergesort(arr, si, mid);
    mergesort(arr, mid + 1, ei);
    merge(arr, si, mid, mid + 1, ei);
  }
}

void merge(int arr[], int fsi, int fei, int ssi, int sei) {
  int mgd[max], x, fh, sh, i;
  fh = fsi;
  sh = ssi;
  x = 0;
  while (fh <= fei && sh <= sei) {
    if (arr[fh] < arr[sh]) {
      mgd[x] = arr[fh];
      x++;
      fh++;
    } else {
      mgd[x] = arr[sh];
      x++;
      sh++;
    }
  }
  while (fh <= fei) {
    mgd[x] = arr[fh];
    x++;
    fh++;
  }
  while (sh <= sei) {
    mgd[x] = arr[sh];
    x++;
    sh++;
  }
  x = 0;
  for (i = fsi; i <= fei; i++) {
    arr[i] = mgd[x];
    x++;
  }
  for (i = ssi; i <= sei; i++) {
    arr[i] = mgd[x];
    x++;
  }
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

Let's assume that the user may not want to sort more than 20 integers, so we define a macro, `max`, of size `20`. Also, an integer array, `arr`, is defined with a size of `max`. You can always increase the size of the `max` macro if required. The user is prompted to enter the numbers that need to be sorted. Suppose the user wants to sort eight integers, so the value 8 that's entered by the user is assigned to the `len` variable. Then, the user is asked to enter the integers to be sorted. The numbers that are entered by the user are assigned to the array, `arr`:



The original unsorted list is displayed on the screen. The `mergesort` function is invoked, and three things are passed to it: the `arr` array; the beginning index location, `0`; and the ending index location, `7`. The arguments that are passed to the function are assigned to the `si` and `ei` parameters.

In the `mergesort` function, we check whether the value of the `ei` variable is more than the `si` variable. If so, the middle value is computed by summing the values of the `si` and `ei` variables and dividing the total by 2, that is, (0+7)/2 . Hence, the value `3` will be assigned to the `mid` variable. Again, the `mergesort` function will be called recursively. In fact, the following three functions will be invoked:

```
mergesort (arr, 0,3)
mergesort(arr, 4,7)
merge(arr,0,3,4,7)
```

When `mergesort(arr, 0, 3)` is called, it checks whether `ei > si` or not. Because 3 > 0, the `mid` variable is computed, *(0+3)/2 = 1*. Again, the following functions will be recursively called:

```
mergesort(arr,0,1)
mergesort(arr,2,3)
merge(arr,0,1,2,3)
```

When `mergesort(arr, 0, 1)` is called, it checks whether `ei > si` or not. Because 1 > 0, the `mid` variable is computed, *(0+1)/2 = 0*. Again, the following functions will be recursively called:

```
mergesort(arr,0,0)
mergesort(arr,1,1)
merge(arr,0,0,1,1)
```

The `mergesort(arr,0,0)` and `mergesort(arr,1,1)` functions will simply return because the `ei > si` condition will not be met. Here, the `merge(arr,0,0,1,1)` function will be invoked. The `merge` function basically merges two arrays comprising one element each. It merges the arrays in such a way that the array appears sorted after they are merged.

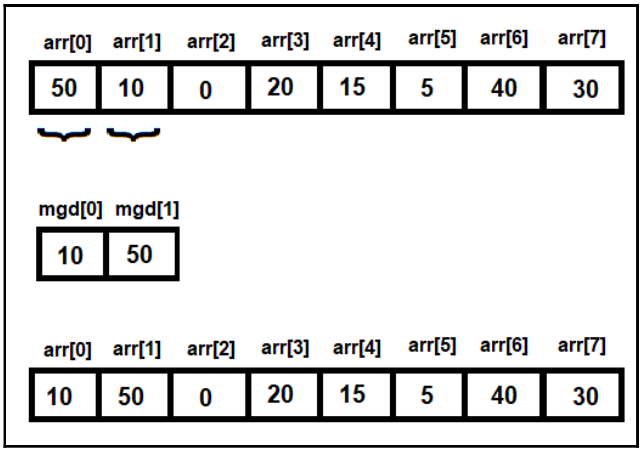
While calling the `merge` function, the `arr`, `0`, `0`, `1`, and `1` arguments will be assigned to the `arr`, `fsi`, `fei`, `ssi`, and `sei` parameters, respectively.

The `fsi` parameter represents the starting index of the first array, `fei` represents the ending index of the first array, `ssi` represents the starting index of the second array, and `sei` represents the ending index of the second array.

While calling the `merge` function, the `arr`, `0`, `0`, `1`, and `1` arguments will be assigned to the `arr`, `fsi`, `fei`, `ssi`, and `sei` parameters, respectively. The values of `fsi` and `ssi`—that is, `0` and `1`, respectively—will be assigned to the two variables, `fh` and `sh`, respectively. `fh` and `sh` represent the index locations of the two arrays to be merged. The sorted and merged numerical will be stored in another array called `mgd`. To represent the index location of the `mgd` array, a variable, `x`, is initialized to `0`.

A `while` loop is set to execute while the index locations of the two arrays are less than their ending indices, that is, while `fh<=fei` and `sh<=sei`, because `0<=0` and `1<=1`, the `while` loop will execute.

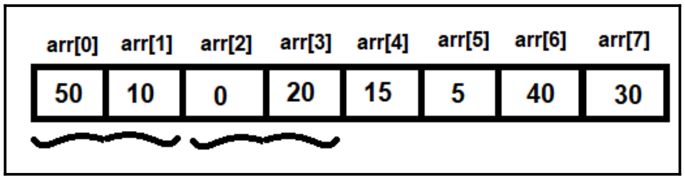
Within the `while` loop, an `if` statement is used to determine which element of the two arrays is smaller. That is, the `if arr[0] < arr[1]` statement is executed to determine whether the first element of the first array is smaller than the first element of the second array. But because 50 is greater than 10, the `else` block will be executed. The value in the `arr[1]` index location will be assigned to the `mgd[0]` location and then the `sh` index will be incremented to 2. The `while` loop will terminate because one of the arrays is over. A `for` loop is set to execute that inserts the element of the `arr[0]` location into the `mgd` array. That is, the `mgd` array will contain the merged and sorted version of the two arrays, `a[0]` and `a[1]`. Finally, a `for` loop is executed to copy the merged array back to the original array, `arr`, as follows:



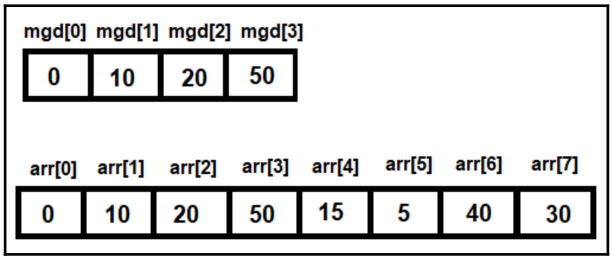
In the second pass, `mergesort(arr,2,3)` is popped from the stack and executed. In recursion, while making the recursive calls, the compiler pushes the statements that couldn't execute because of the recursion to stack so that they can be popped from the stack later and executed.

The `mergesort(arr,2,3)` statement will sort and merge the `arr[2]` and `arr[3]` elements. Because `arr[2]` is already less than `arr[3]`, no interchanging of elements will take place.

Thereafter, the `merge(arr,0,1,2,3)` statement will be popped out from the stack and is executed. This statement will make the `arr` array appear as two subarrays: the first subarray will consist of the `arr[0]` and `arr[1]` elements, while the second subarray of the `arr[2]` and `arr[3]` elements and will appear as follows:



Now, these two subarrays are sorted and merged into a temporary array, `mgd`, and then transferred to the `arr` array, as follows:

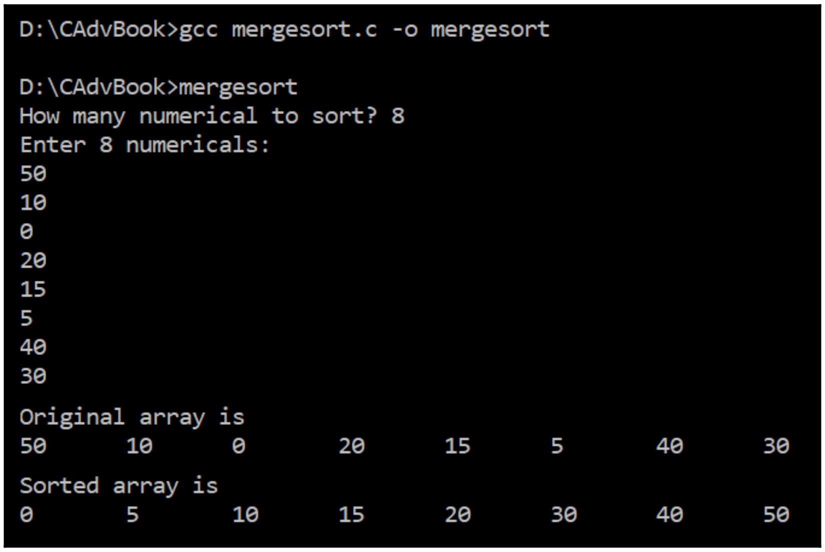


mgd[0] mgd[1] mgd[2] mgd[3]
0 10 20 50
arr[0] arr[1] arr[2] arr[3] arr[4] arr[5] arr[6] arr[7]
0 10 20 50 15 5 40 30

This procedure is repeated for the rest of the elements of the array. The program is compiled using GCC with the following statement:

```
gcc mergesort.c –o mergesort
```

Because no error appears upon compilation, this means the `mergesort.c` program has successfully compiled into the `mergesort.exe` file. On executing the file, it will ask you to specify how many numbers need to be sorted. Following this, the program prompts the user to enter the numbers to be sorted. After entering the numbers, they will appear sorted in ascending order, as shown in the following screenshot:



```
D:\CAdvBook>gcc mergesort.c -o mergesort

D:\CAdvBook>mergesort
How many numerical to sort? 8
Enter 8 numericals:
50
10
0
20
15
5
40
30

Original array is
50      10      0       20      15      5       40      30

Sorted array is
0       5       10      15      20      30      40      50
```

Now, let's move on to the next recipe!

Arranging numbers in ascending order using shell sort

In this recipe, we will learn how to arrange some integers in ascending order using the shell sort technique.

How to do it...

Basically, shell sort separates an array into subarrays, where the subarrays contain every *k*th element of the original array. The value of `k` is called an increment. The increment value can be assumed to be any value. Suppose there is an array, `arr`, of 10 elements. If that's the case, the following steps will help sort those array elements in ascending order using the shell short technique:

1. Let's assume the value of the increment is `5`. Here, the following subarrays will be made:

   ```
   arr[0], arr[5]
   arr[1], arr[6]
   arr[2], arr[7]
   arr[3], arr[8]
   arr[4], arr[9]
   ```

   Each of the subarrays is individually sorted.

2. Once all of the subarrays have been sorted, a new, smaller value for the increment is chosen and the array is partitioned into a new set of subarrays. For example, if the increment is reduced to the value `3`, the subarrays will now be as follows:

   ```
   arr[0], arr[3], arr[6], arr[9]
   arr[1], arr[4], arr[7]
   arr[2], arr[5], arr[8]
   ```

3. Each of the preceding subarrays will be sorted and the process is repeated with an even smaller value of *k*.
4. Eventually, the value of *k* is set to 1 so that the subarray consisting of the entire array is sorted. A decreasing sequence of increments is fixed at the start of the entire process. The last value in this sequence must be 1.

On the last iteration, when the value of the increment is equal to `1`, the sort reduces to a simple insertion sort. The simple insertion sort works very efficiently on arrays that are in an almost sorted order.

The program for sorting the elements of an integer array using the shell sort technique is as follows:

```
//shellsort.c

#include <stdio.h>

#
define max 20
int main() {
  int len, i, arr[max], incr, j, temp;

  printf("How many values to be sorted ?");
  scanf("%d", & len);
  printf("Enter %d values\n", len);
  for (i = 0; i < len; i++)
    scanf("%d", & arr[i]);
  printf("\nOriginal array is\n");
  for (i = 0; i < len; i++)
    printf("%d\t", arr[i]);
  printf("\n");
  incr = 5;
  while (incr >= 1) {
    for (i = 0; i < len - incr; i++) {
      for (j = i + incr; j < len; j += incr) {
        if (arr[i] > arr[j]) {
          temp = arr[i];
          arr[i] = arr[j];
          arr[j] = temp;
        }
      }
    }
    printf("\nThe array when increment is %d\n", incr);
    for (i = 0; i < len; i++) printf("%d\t", arr[i]);
    printf("\n");
    incr = incr - 2;
  }
  return 0;
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

A macro is defined by the name `max` and is set to a size of `20`, assuming that the user may not want to sort more than 20 values. You can always increase the value of the `max` macro. An integer array, `arr`, is defined as a size of `max`. The user is prompted to specify how many numbers need to be sorted. Suppose the user wants to sort eight integers. The value `8` that's entered by the user is assigned to the `len` variable. Then, the user is prompted to enter the values to be sorted, which are assigned to the `arr` array, as follows:

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]
11	4	9	2	6	15	10	3

In the shell sort method, an array is sorted in the form of increments. The value of the `incr` variable is set to `5`. Because the increment value, `incr`, is `5`, all of the *n*th elements are compared with its `n+5`th element and an insertion sort is performed between them. That is, `arr[0]` is compared with `arr[5]`; `arr[1]` is compared with `arr[6]`; `arr[2]` is compared with `arr[7]`; and the insertion sort is performed among these pairs:

- `arr[0]`, `arr[5]`: 11, 15 will remain unchanged
- `arr[1]`, `arr[6]`: 4, 10 will remain unchanged
- `arr[2]`, `arr[7]`: 9, 3 is sorted as 3, 9

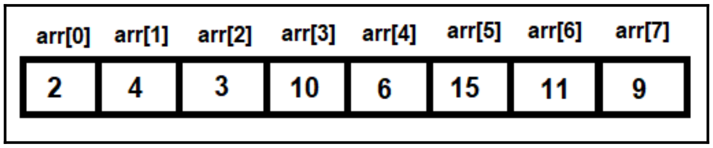
After the first pass, the `arr` array will appear as follows:

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]
11	4	3	2	6	15	10	9

The increment is decreased by 2 after every pass. So, now, the `incr` increment value has become `3`. Afterward, arr[0] is compared with `arr[3]`; `arr[1]` is compared with `arr[4]`; `arr[2]` is compared with `arr[5]`; `arr[3]` is compared with `arr[6]`; `arr[4]` is compared with `arr[7]`; and the insertion sort is performed among these pairs as listed here:

- `arr[0]`, `arr[3]`, `arr[6]`: 1, 2, 10 is sorted as 2, 10, 11.
- `arr[1]`, `arr[4]`, `arr[7]`: 4, 6, 9 is already sorted, so there are no changes in this sequence.
- `arr[2]`, `arr[5]`: 3, 15 is already sorted, so there are no changes in this sequence.

The result of the second pass is as follows:

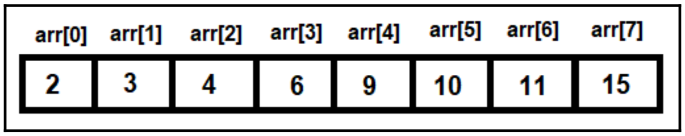


Again, the increment is decreased by 2. The increment value was `3`, so now the `incr` increment value has become `1`. Now, `arr[0]` is compared with `arr[1]`; `arr[1]` is compared with `arr[2]`; `arr[2]` is compared with `arr[3]`; `arr[3]` is compared with `arr[4]`; `arr[4]` is compared with `arr[5]`; and so on. Also, the insertion sort is performed among the following pairs:

- `arr[0]`,`arr[1]`: 2, 4 is already sorted, so there are no changes in this sequence.
- `arr[1]`, `arr[2]`: 4, 3 is sorted as 3, 4. To perform insertion sort, `arr[0]` and `arr[1]` are compared to keep the smallest value at the `arr[0]` index location.
- `arr[2]`,`arr[3]`: 4, 10 is already sorted, so there are no changes in this sequence.
- `arr[3]`,`arr[4]`: 10, 6 is sorted as 6, 10. Insertion sort will be applied at the earlier index locations,that is, `arr[2]` and `arr[3]`, `arr[1]` and `arr[2]`, and `arr[0]` and `arr[1]`. Values at these index locations are compared and the tendency is to keep them sorted.
- `arr[4]`,`arr[5]`: 10, 15 is already sorted, so there are no changes in this sequence.

- `arr[5]`,`arr[6]`: 15, 11 is sorted as 11, 15. After the interchanging of values at the `arr[5]` and `arr[6]` index locations and the `arr[4]` and `arr[5]` index locations, `arr[3]` and `arr[4]`, `arr[2]` and `arr[3]`, `arr[1]` and `arr[2]`, and `arr[0]` and `arr[1]` are also compared and interchanging is performed to keep them sorted.
- `arr[6]`,`arr[7]`: 15, 9 is sorted as 9, 15. After the interchanging of values at the `arr[6]` and `arr[7]` index locations, insertion sort is applied at the `arr[5]`, `arr[4]`, `arr[3]`, `arr[2]`, `arr[1]`, and `arr[0]` index locations too.

The result of the third pass is as follows:



A `while` loop is set to execute while the value of the `incr` variable is greater than or equal to 1. Within the `while` loop, a `for` loop is executed whose value ranges from 0 until 8-5 three times.

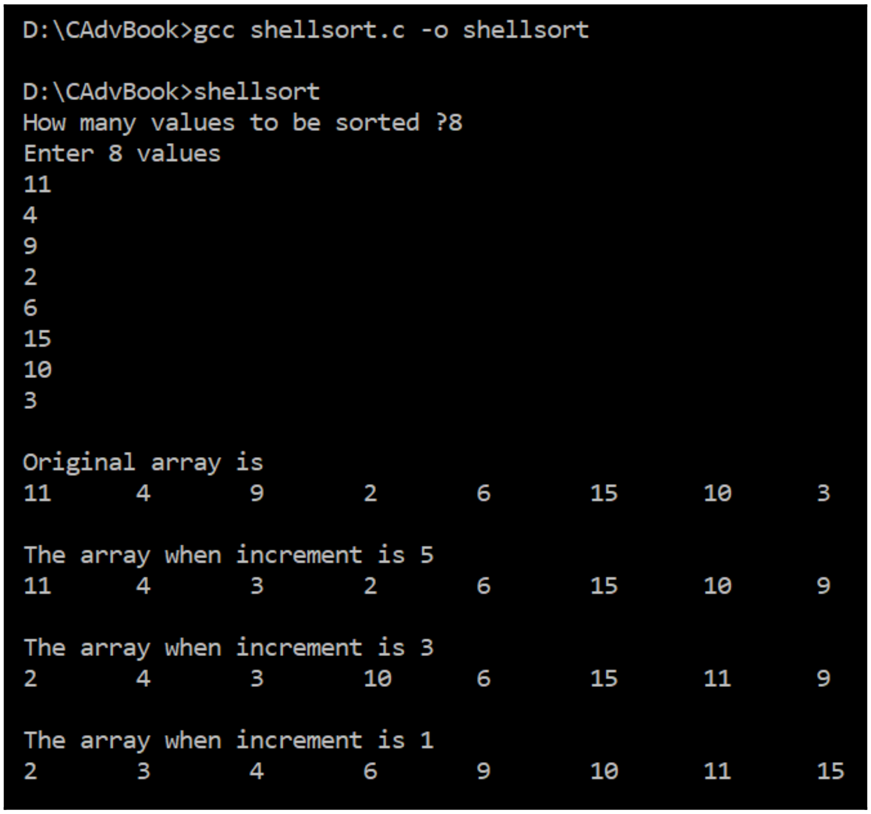
Within the outer `for` loop, one nested `for` loop is also used where the value of `j` will range from `for (j=0+5; j<8;j+=5)`. The value of `j` will start at `5`.

The idea is to compare `arr[0]` with `arr[5]`. An `if` statement is invoked with the following index locations: `if( arr[0] >arr[5])`. Then, the values at these index locations have to be interchanged. The values at the `arr[0]` and `arr[5]` index locations are 11 and 15, respectively. Because 11 < 15, no interchanging will happen and the value of `j` is incremented to 10, which is not less than `len` – that is, 8 – so the inner `for` loop will terminate. The outer `for` loop will increment the value of the `i` variable to 1. The inner for `j` loop will run from `for (j=1+5;j<8;i+=5)`, that is, `arr[1]` and `arr[6]` are compared. Because 4 is already less than 6, no interchanging will take place.

The program is compiled using GCC with the following statement:

```
gcc shellsort.c -o shellsort
```

Because no error appears upon compilation, that means the `shellsort.c` program has successfully compiled into the `shellsort.exe` file. On executing the file, it will ask the user to specify how many numbers are to be sorted. Following this, the program prompts the user to enter the numbers to be sorted. After entering the numbers, they will appear sorted as per the level of increment. The final sorted list is the one where the value of the increment is `1`, as shown in the following screenshot:



```
D:\CAdvBook>gcc shellsort.c -o shellsort

D:\CAdvBook>shellsort
How many values to be sorted ?8
Enter 8 values
11
4
9
2
6
15
10
3

Original array is
11      4       9       2       6       15      10      3

The array when increment is 5
11      4       3       2       6       15      10      9

The array when increment is 3
2       4       3       10      6       15      11      9

The array when increment is 1
2       3       4       6       9       10      11      15
```

Now, let's move on to the next recipe!

Arranging numbers in ascending order using radix sort

In this recipe, we will learn how to sort certain numbers in ascending order using the radix sort technique.

How to do it...

Follow these steps to learn how to sort numerical values using radix sort:

1. Prompt the user to enter array elements to be sorted.
2. Ask the user for the number of digits.
3. Use the `mod` operator to find the unit value of each number. Sort all of the numbers based on the units and put them into a temporary array.
4. Get all of the numbers from the temporary array and use the `mod` operator to get the tenth unit of all the numbers. Sort all of the numbers based on the tenth unit and put them back into the temporary array.
5. Get all of the numbers from the temporary array and use the `mod` operator to get the hundredth unit of all the numbers. Sort all of the numbers based on the hundredth unit and place them in the temporary array.
6. The temporary array will have the final sorted order after the third pass we mentioned previously. This is because the maximum number of digits in the numerical was 3, so the final sorted order will be received after the third pass.

The program for sorting the elements of an integer array using the radix sort technique is as follows:

```
//radixsort.c

#include<stdio.h>

#include <math.h>

#
define max 20
int main() {
  int arr[max], k, e, len, i, j, digit, x, numbdig;
  int temp[10][max];
  printf("How many numbers to sort ? ");
  scanf("%d", & len);
```

```
    printf("How many digits in each number ? ");
    scanf("%d", & numbdig);
    printf("Enter %d values\n", len);
    for (i = 0; i < len; i++)
      scanf("%d", & arr[i]);
    for (i = 0; i < 10; i++) {
      for (j = 0; j < max; j++) {
        temp[i][j] = -1;
      }
    }
    for (k = 1; k <= numbdig; k++) {
      e = pow(10, k - 1);
      for (i = 0; i < len; i++) {
        digit = (arr[i] / e) % 10;
        x = 0;
        while (temp[digit][x] != -1) x++;
        temp[digit][x] = arr[i];
      }
      x = 0;
      for (i = 0; i < 10; i++) {
        for (j = 0; j < max; j++) {
          if (temp[i][j] != -1) {
            arr[x] = temp[i][j];
            x++;
          }
        }
      }
      printf("\nThe output of pass %d is \n", k);
      for (i = 0; i < len; i++) printf("%d\t", arr[i]);
      printf("\n");
      for (i = 0; i < 10; i++) {
        for (j = 0; j < max; j++) {
          temp[i][j] = -1;
        }
      }
    }
    return 0;
  }
```

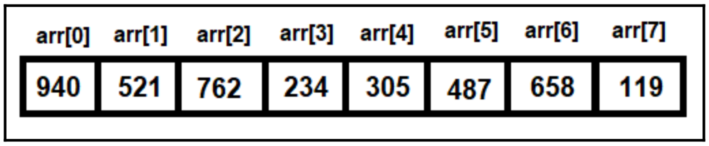
Now, let's go behind the scenes to understand the code better.

How it works...

In radix sort, sorting requires several passes, where the number of passes equals the maximum number of digits in the supplied numbers.

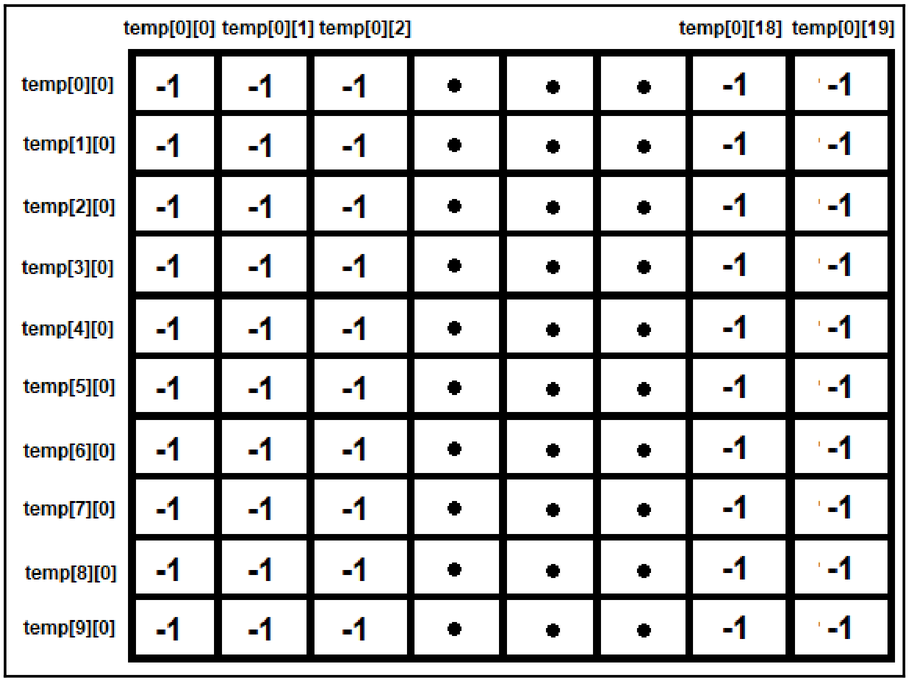
For example, if the numerical data being sorted consists of three digits, the radix sort requires three passes to sort them. Sorting begins by arranging the numerical data based on the **Least Significant Digit** (**LSD**) and with every pass, we move toward the **Most Significant Digit** (**MSD**). We begin with the LSD and store the numerical data in a two-dimensional array in the ascending order of the LSD. In the second pass, the numerical data (that was sorted on the LSD) is placed in the two-dimensional array, but this time, in ascending order of the next significant digit. In this way, with every pass, we move toward the MSD. The last pass consists of storing the numerical data in the ascending order of the MSD in the two-dimensional array.

For example, let's assume the numerical data to be sorted is as follows:



arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]
940	521	762	234	305	487	658	119

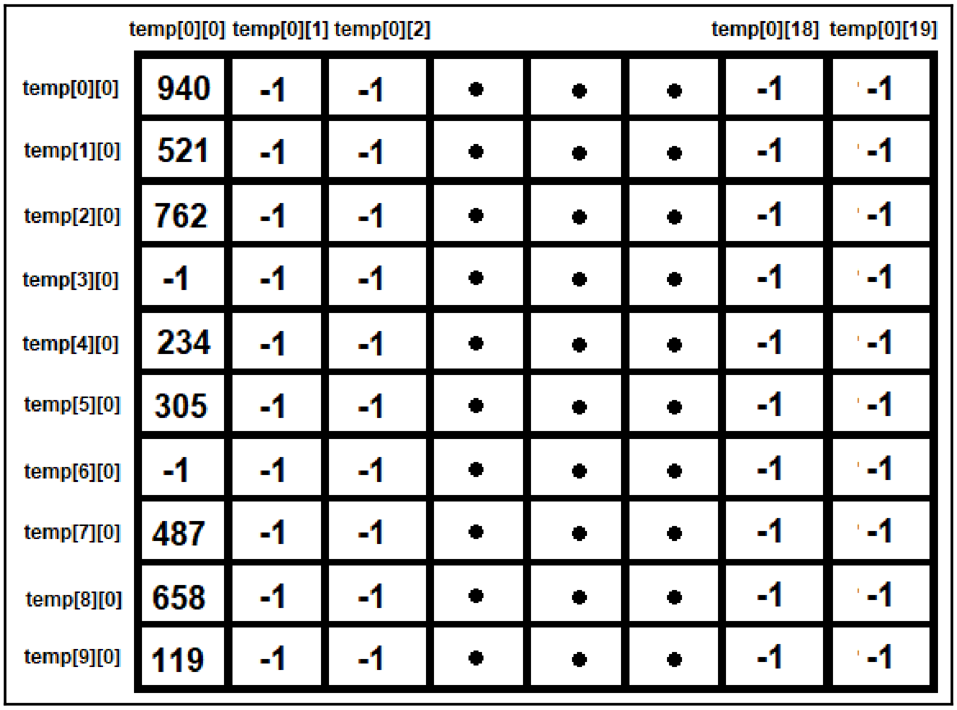
A temporary two-dimensional array called **temp** is defined and all of its elements are initialized to **-1**, as shown in the following screenshot. The cell with the value **-1** will indicate that the cell is empty and hasn't been assigned a value yet:



	temp[0][0]	temp[0][1]	temp[0][2]				temp[0][18]	temp[0][19]
temp[0][0]	-1	-1	-1	●	●	●	-1	-1
temp[1][0]	-1	-1	-1	●	●	●	-1	-1
temp[2][0]	-1	-1	-1	●	●	●	-1	-1
temp[3][0]	-1	-1	-1	●	●	●	-1	-1
temp[4][0]	-1	-1	-1	●	●	●	-1	-1
temp[5][0]	-1	-1	-1	●	●	●	-1	-1
temp[6][0]	-1	-1	-1	●	●	●	-1	-1
temp[7][0]	-1	-1	-1	●	●	●	-1	-1
temp[8][0]	-1	-1	-1	●	●	●	-1	-1
temp[9][0]	-1	-1	-1	●	●	●	-1	-1

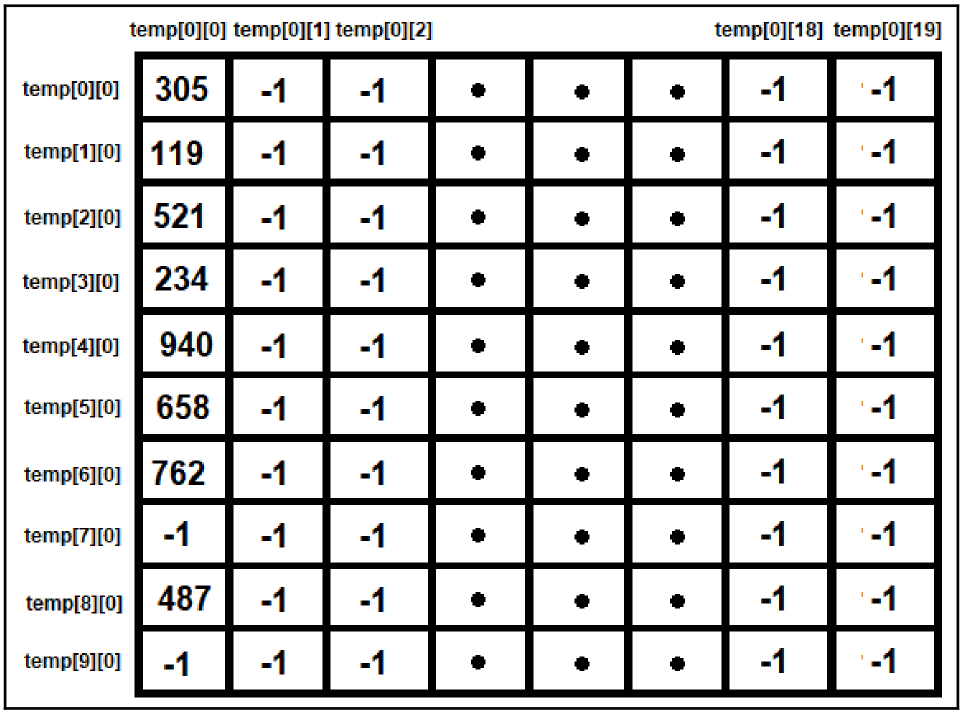
The important thing to note here is that the size of all of the numbers is the same, that is, all of the numbers are three digits in size. If a number has fewer digits, it is padded with 0s. For example, if a number is 59, it is written as 059 to make it three digits long. Now, since the data here is all made up of three digits, there are three passes to sort this data.

In the first pass, we store the data in a two-dimensional array based on the LSD. This means that all of the numbers with the LSD as 0 are placed in the 0th row, all of the numbers with the LSD as 1 are placed in the 1st row, and so on, as follows:



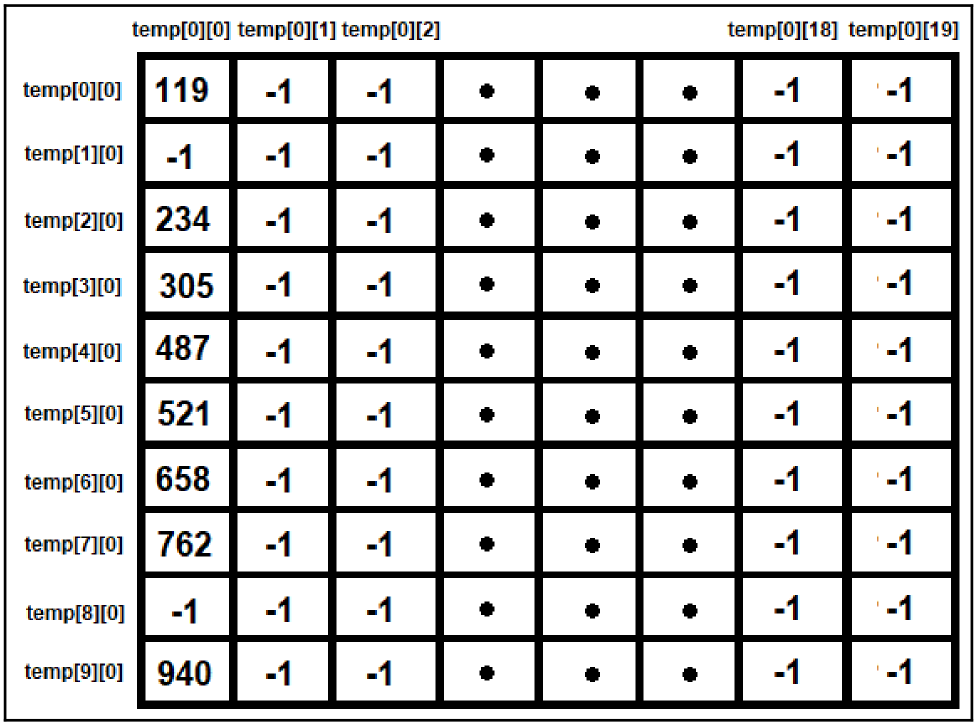
	temp[0][0]	temp[0][1]	temp[0][2]				temp[0][18]	temp[0][19]
temp[0][0]	940	-1	-1	●	●	●	-1	-1
temp[1][0]	521	-1	-1	●	●	●	-1	-1
temp[2][0]	762	-1	-1	●	●	●	-1	-1
temp[3][0]	-1	-1	-1	●	●	●	-1	-1
temp[4][0]	234	-1	-1	●	●	●	-1	-1
temp[5][0]	305	-1	-1	●	●	●	-1	-1
temp[6][0]	-1	-1	-1	●	●	●	-1	-1
temp[7][0]	487	-1	-1	●	●	●	-1	-1
temp[8][0]	658	-1	-1	●	●	●	-1	-1
temp[9][0]	119	-1	-1	●	●	●	-1	-1

The numerical data that's been arranged based on the LSD is taken out of the array. The sequence of data is 940, 521, 762, 234, 305, 487, 658, 119. This data is then placed in the array based on the middle digit. That is, all of the numbers with the middle digit as 0 are placed in the 0th row, all of the numbers with the middle digit as 1 are placed in the 1st row, and so on, as follows:



	temp[0][0]	temp[0][1]	temp[0][2]				temp[0][18]	temp[0][19]
temp[0][0]	305	-1	-1	●	●	●	-1	-1
temp[1][0]	119	-1	-1	●	●	●	-1	-1
temp[2][0]	521	-1	-1	●	●	●	-1	-1
temp[3][0]	234	-1	-1	●	●	●	-1	-1
temp[4][0]	940	-1	-1	●	●	●	-1	-1
temp[5][0]	658	-1	-1	●	●	●	-1	-1
temp[6][0]	762	-1	-1	●	●	●	-1	-1
temp[7][0]	-1	-1	-1	●	●	●	-1	-1
temp[8][0]	487	-1	-1	●	●	●	-1	-1
temp[9][0]	-1	-1	-1	●	●	●	-1	-1

The numerical data that's arranged based on the middle digit is taken out of the array. The sequence of data will be 305, 119, 521, 234, 940, 658, 762, 487. This data is then placed in the array based on the MSD. That is, all of the numbers with the MSD as 0 are placed in the 0th row, all of the numbers with the MSD as 1 are placed in the 1st row, and so on, as follows:

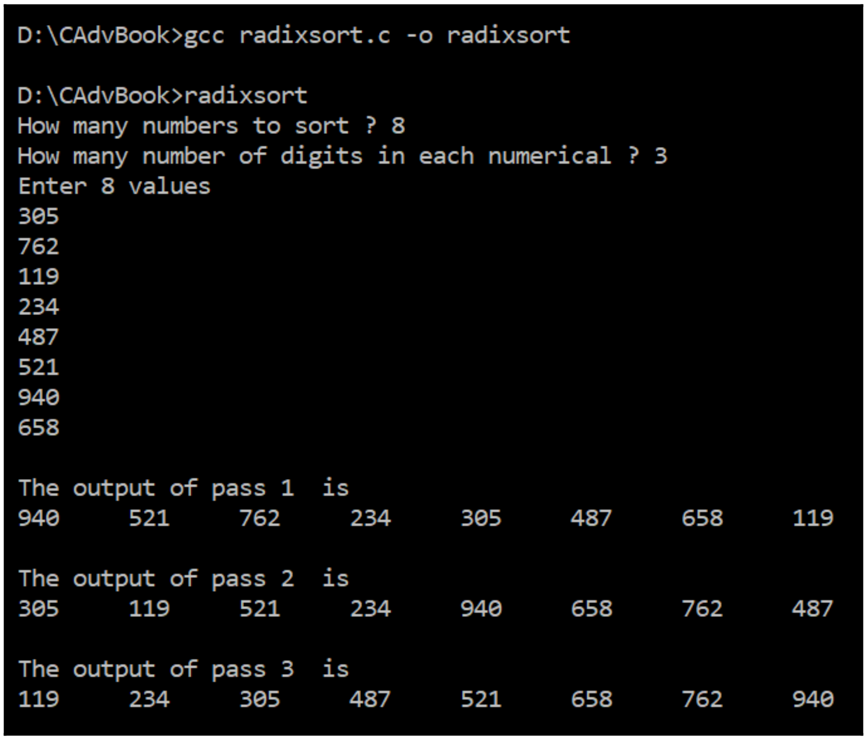


	temp[0][0]	temp[0][1]	temp[0][2]				temp[0][18]	temp[0][19]
temp[0][0]	119	-1	-1	●	●	●	-1	-1
temp[1][0]	-1	-1	-1	●	●	●	-1	-1
temp[2][0]	234	-1	-1	●	●	●	-1	-1
temp[3][0]	305	-1	-1	●	●	●	-1	-1
temp[4][0]	487	-1	-1	●	●	●	-1	-1
temp[5][0]	521	-1	-1	●	●	●	-1	-1
temp[6][0]	658	-1	-1	●	●	●	-1	-1
temp[7][0]	762	-1	-1	●	●	●	-1	-1
temp[8][0]	-1	-1	-1	●	●	●	-1	-1
temp[9][0]	940	-1	-1	●	●	●	-1	-1

The program is compiled using GCC with the following statement:

```
gcc radixsort.c -o radixsort
```

Because no error appears upon compilation, this means the `radixsort.c` program has successfully compiled into the `radixsort.exe` file. On executing the file, it will ask the user to specify how many numbers need to be sorted. Following this, the program prompts the user to enter the numbers to be sorted. After entering the numbers, they will appear sorted in ascending order. The output will appear in different passes, depending on the maximum number of digits in the supplied number. Because the number of digits in the supplied number is 3, the output will appear in three passes and the output of pass three is the final sorted version of the supplied number, as shown in the following screenshot:



```
D:\CAdvBook>gcc radixsort.c -o radixsort

D:\CAdvBook>radixsort
How many numbers to sort ? 8
How many number of digits in each numerical ? 3
Enter 8 values
305
762
119
234
487
521
940
658

The output of pass 1  is
940     521     762     234     305     487     658     119

The output of pass 2  is
305     119     521     234     940     658     762     487

The output of pass 3  is
119     234     305     487     521     658     762     940
```