

Emphasis on: Two-dimensional array processing

A radix sort (commonly known as a bucket sort) of integers begins with a one-dimensional array of integers to be sorted, and a two-dimensional array of integers with 10 rows and n columns, where n is the number of values in the one-dimensional array to be sorted. Each row of the two-dimensional array is known as a bucket.

Alternatively you could use an array with n rows and 10 columns (buckets), though that is more difficult to program.

Write a function to bucket sort an array of integers and a program to test your function.

The sort algorithm is as follows:

- 1) Loop through the one-dimensional array and place each of its values in the bucket whose subscript corresponds to the value of its one's digit. For example, 97 is placed in bucket 7, 3 is placed in bucket 3, 140 is placed in bucket 0, 47 is placed in bucket 7, and 93 is placed in bucket 3.
- 2) Loop through the buckets and copy the values back to the original array (from bucket 0 to bucket 9, and from first value entered into the bucket to last). The new order of the example values in the one-dimensional array is 140, 3, 93, 97, 47.
- 3) Repeat this process for each subsequent digit position (tens, hundreds, etc.) until all possible digit positions have been processed.

On the second pass of the array (looking at the tens position), 140 is placed in bucket 4, 3 in bucket 0, 93 in bucket 9, 97 in bucket 9, and 47 in bucket 4. The order of the values in the one-dimensional array is now 3, 140, 47, 93, 97.

On the third pass, 3 is placed in bucket 0, 140 in bucket 1, 47 in bucket 0, 93 in bucket 0, and 97 in bucket 0. The bucket sort is guaranteed to have all the values properly sorted after processing the leftmost digit of the largest number. The bucket sort knows it is done when all the values are copied into bucket 0.

The bucket sort requires one pass through the array for each possible digit in the numbers to be sorted. If you don't know the maximum number of digits in your largest number, you'd have to make another pass. When all the numbers are in the 0 bucket, you know they are sorted.

Note that the memory required for the ten buckets is ten times the size of the array being sorted. This sorting technique provides better performance in terms of speed than many sorts, but requires much larger data storage capacity. Bucket sort is an example of a time-space tradeoff. It performs faster than many sorts, but uses considerably more memory.

You are to use a random number generator to fill an array with integers in the range 0 to 32,000, Inclusive. This array should be passed to your Bucket Sort function to be sorted into ascending order. Print the values in their original order first and then again after each pass through the array, ending with the values in their sorted order.

Program Requirements:

1. The bucket sort must be written as a function. Parameters passed are the one-dimensional array to be sorted and the number of elements in the array.
2. The bucket array must be implemented as a two-dimensional array.
3. Test your function with two different random sequences of integers, one of them a sequence of 25 integers and the other a sequence of 20 integers.
4. Your sort should not depend on knowing the maximum number of digits in the values to be sorted (test for all values in the 0 bucket).

Random Number Generator

The standard library function `rand()` returns a random number such that $0 \leq \text{rand}() < 32767$.

To convert this into a number within a specific range $1 \dots n$, use the algorithm `rand() % n + 1`.

For instance, if we were simulating rolling dice, we would want a number in the range $1 \dots 6$, so we could code something like: `die = rand() % 6 + 1`.

To obtain a number within the range $0 \dots n$, use the algorithm `rand() % (n + 1)`.

The random number generator is initialized (needs to be called only once) by a call to the standard library function `srand()`. You can pass `srand` a specific value (good for testing purposes until you're sure your program is working correctly), or you can initialize it with a value from the system clock (for a random starting seed):

`srand (237);` or `srand (unsigned (time (NULL)));`

To use the `rand` and `srand` functions, include `stdlib.h` (or `cstdlib`), and to use the `time` function, include `time.h` (or `ctime`).

Here's another example sorting these numbers: 25 150 53 254 300 4 355 404 523 225 80
Buckets after the first pass:

	0	1	2	3	4	5	6	7	...
0	150	300	80						
1									
2									
3	53	523							
4	254	4	404						
5	25	355	225						
6									
7									
8									
9									

Emptying out the buckets, we have: 150 300 80 53 523 254 4 404 25 355 225
Buckets after the second pass:

	0	1	2	3	4	5	6	7	...
0	300	4	404						
1									
2	523	25	225						
3									
4									
5	150	53	254	355					
6									
7									
8	80								
9									

Emptying out the buckets again, we have: 300 4 404 523 25 225 150 53 254 355 80
Buckets after the third pass:

	0	1	2	3	4	5	6	7	...
0	4	25	53	80					
1	150								
2	225	254							
3	300	355							
4	404								
5	523								
6									
7									
8									
9									

Emptying out the buckets again, we have: 4 25 53 80 150 225 254 300 355 404 523
After the next pass, all the values are sorted into the 0 row (bucket) and the other buckets are empty.