1 What it Does

Sorts a list of (preferably small) nonnegative integers for which we know the maximum. This can be tweaked to do a bit more.

2 How it Works

Counting sort works by counting the number of each integer and creating a new list with that information. For example consider this list:

A \[ 3 \ 10 \ 4 \ 3 \ 4 \ 5 \ 4 \ 3 \ 5 \ 6 \ 1 \ 3 \ 0 \]

Knowing that the maximum value is 10 we create an array LOC with indices from 0 to 10 such that LOC[k] contains the count of k in A. The name LOC we're using may seem a bit weird but it will make sense soon.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

# of 0s 1s 2s 3s 4s 5s 6s 7s 8s 9s 10s

Next we modify LOC above by replace it with its cumulative version, meaning we successively add each entry to the next.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

For a moment observe that this needs to give us the sorted version of A which looks like this:

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

Observe that each LOC[i]==j indicates that in the sorted version of the array if there are any occurrences of i then they end at location j, meaning index j-1.

For example LOC[3]==6 which means that in the sorted array the 3s end at location 6, meaning index 5, and LOC[4]==9 which means that in the sorted array the 4s end at location 9, meaning index 8.

Then we construct a new list ANEW by going through A in reverse order, taking each element, finding the index to place it using LOC, then decreasing that entry in LOC in preparation in case there’s another value to come. So think about A as the input list, LOC as the processing list, and ANEW as the output list.
Here are the first two steps:

\( \mathbf{A}[12] = 0 \) (the value) and \( \mathbf{LOC}[0] = 1 \) (the location, one more than the index, to put it) so assign \( \mathbf{ANEW}[1-1] = 0 \)...

\[
\begin{array}{cccccccccccc}
A & 3 & 10 & 4 & 3 & 4 & 5 & 4 & 3 & 5 & 6 & 1 & 3 & 0 \\
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{LOC} & 1 & 2 & 2 & 6 & 9 & 11 & 12 & 12 & 12 & 12 & 13 \\
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\text{ANEW} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

...and update \( \mathbf{LOC}[0] = 0 \) (ready for the next 0, which there isn’t one).

\( \mathbf{A}[11] = 3 \) (the value) and \( \mathbf{LOC}[3] = 6 \) (the location, one more than the index, to put it) so assign \( \mathbf{ANEW}[6-1] = 3 \)...

\[
\begin{array}{cccccccccccc}
A & 3 & 10 & 4 & 3 & 4 & 5 & 4 & 3 & 5 & 6 & 1 & 3 & 0 \\
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{LOC} & 1 & 2 & 2 & 6 & 9 & 11 & 12 & 12 & 12 & 12 & 13 \\
\text{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\text{ANEW} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\
\end{array}
\]

...and update \( \mathbf{LOC}[3] = 5 \) (ready for the next 3, which there is one).

And so on until we have filled up \( \mathbf{ANEW} \). Typically we then copy \( \mathbf{ANEW} \) back to \( \mathbf{A} \).

The reason for going through \( \mathbf{A} \) in reverse order is that it results in CountingSort being stable.
3 Pseudocode and Time Complexity

Here is the pseudocode with time assignments. Everything is just assignment, notice not a single comparison in sight!

\PRE: A is a list of length n with maximum value k.
LOC = array of zeros of length k+1 \( c_2(k+1) \)
ANEW = array of zeros of length n \( c_2n \)
for i = 0 to n-1
    LOC[A[i]] = LOC[A[i]] + 1 \( c_1 \)
end
for i = 1 to k
    LOC[i] = LOC[i] + LOC[i-1] \( c_1 \)
end
for i = n-1 down to 0
    ANEW[LOC[A[i]]-1] = A[i] \( c_1 \)
    LOC[A[i]] = LOC[A[i]] - 1 \( c_1 \)
end
for i = 0 to n-1
    A[i] = ANEW[i] \( c_1 \)
end
\POST: A is sorted.

Note 3.0.1. If A is given but k is not we can simply add a simple loop to calculate it first since it’s the maximum of the list. This process is \( \Theta(n) \).

Best-, Worst-, and Average-Cases:
In most languages it takes time equal to the length of a list to allocate an empty list, or a list of 0s, so we start with \( c_2(k+1) \) and \( c_2n \). In addition then we have \( c_1n \) for the first loop, \( c_1k \) for the second loop, \( 2c_1n \) for the third loop, and \( c_1n \) for the fourth loop.

The total time required is then:

\[
T(n, k) = c_2(k+1) + c_2n + 2c_1 + c_1n + c_1k + 2c_1n + c_1n = \Theta(n + k)
\]

This is the same for best-case, worst-case, and average-case.
If this is confusing, think of \( n + k \) as a single variable, so what this means is that there are \( B, C > 0 \) and some index \( m_0 \) such that:

If \( n + k \geq m_0 \) then \( B(n + k) \leq T(n, k) \leq C(n + k) \)

Note 3.0.2. If k is a fixed constant and we let n vary then the time complexity is \( \Theta(n) \).

Note 3.0.3. If k is not fixed but we can guarantee that \( k \leq n \) then the time complexity is \( \Theta(n) \).
4 Auxiliary Space

CountingSort uses $\Theta(n + k)$ auxiliary space since it is required to create the LOC list as well as the ANEW list.

5 Stability

Our particular pseudocode implementation of CountingSort is stable.

6 In-Place

CountingSort is not in-place.

7 Usage Note

CountingSort can be modified to manage other types of data. See the exercises for examples. It is often used as an auxiliary method inside other methods like RadixSort.
8 Thoughts, Problems, Ideas

1. Suppose \( k_1, k_2 \in \mathbb{Z} \) with \( k_1 < k_2 \). Suppose every \( x \) in your list is an integer satisfying \( k_1 \leq x \leq k_2 \). Modify CountingSort so that it can manage this list. What would the \( \Theta \) time complexity be?

2. Suppose \( k_1, k_2 \in \mathbb{R} \) with \( k_1 < k_2 \) and both having at most one digit after the decimal point. Suppose every \( x \) in your list is a real number satisfying \( k_1 \leq x \leq k_2 \) with \( x \) having at most one digit after the decimal point. Modify CountingSort so that it can manage this list. What would the \( \Theta \) time complexity be?

3. Suppose \( d \in \mathbb{Z} \) with \( d \geq 0 \), \( k_1, k_2 \in \mathbb{R} \) with \( k_1 < k_2 \) and both having at most \( d \) digits after the decimal point. Suppose every \( x \) in your list is a real number satisfying \( k_1 \leq x \leq k_2 \) with \( x \) having at most \( d \) digits after the decimal point. Modify CountingSort so that it can manage this list. What would the \( \Theta \) time complexity be?

4. Using the list \([2, 2', 2'', 1]\) demonstrate why our implementation of CountingSort is unstable. Understanding the flow of the algorithm with this example should indicate to you how CountingSort can be very simply modified to make it stable. What would this modification be?
import random
A = []
k = 5
n = 10
for i in range(0,n):
    A.append(random.randint(0,k))
print(A)
ANEW = [0] * n
LOC = [0] * (k+1)
for i in range(0,n):
    LOC[A[i]] = LOC[A[i]] + 1
print('Count array: '+str(LOC))
for i in range(1,k+1):
    LOC[i] = LOC[i] + LOC[i-1]
print('Cumulative count array: '+str(LOC))
for i in range(n-1,-1,-1):
    LOC[A[i]] = LOC[A[i]] - 1
    print('Positioning A['+str(i)+'] in position '+str(LOC[A[i]]-1))
    print('Result: '+str(ANEW))
for i in range(0,n):
    A[i] = ANEW[i]
print(A)
Output:

[3, 2, 0, 5, 2, 5, 0, 1, 2, 1]
Count array: [2, 2, 3, 1, 0, 2]
Cumulative count array: [2, 4, 7, 8, 8, 10]
Result: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Result: [0, 0, 0, 1, 0, 0, 2, 0, 0, 0]
Positioning A[7] in position 1
Result: [0, 0, 1, 1, 0, 2, 0, 0, 0, 0]
Result: [0, 0, 1, 1, 0, 2, 0, 0, 0, 0]
Result: [0, 0, 1, 1, 0, 2, 0, 0, 0, 5]
Result: [0, 0, 1, 1, 0, 2, 2, 0, 0, 5]
Result: [0, 0, 1, 1, 0, 2, 2, 0, 5, 5]
Result: [0, 0, 1, 1, 0, 2, 2, 0, 5, 5]
Result: [0, 0, 1, 1, 2, 2, 2, 0, 5, 5]
Positioning A[0] in position 6
Result: [0, 0, 1, 1, 2, 2, 2, 3, 5, 5]

[0, 0, 1, 1, 2, 2, 2, 3, 5, 5]