Sort Stability

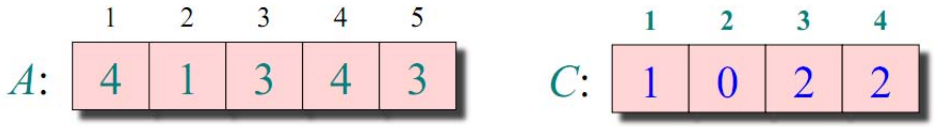
A sorting algorithm is **stable** if elements with the same key appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array. Normally, the property of stability is important when satellite data are carried around with the element being sorted. For example, in order for radix sort to work correctly, the digit sorts must be stable.

Counting Sort

Counting sort is an algorithm that takes an array A of n elements with keys in the range {1, 2, ..., k} and sorts the array in $O(n + k)$ time. It is a stable sort.

In the lecture, we have seen one implementation of counting sort. Here we will show another one mentioned in the text book (CLRS).

Intuition: Count key occurrences using an auxiliary array C with k elements, all initialized to 0. We make one pass through the input array A, and for each element i in A that we see, we increment $C[i]$ by 1. After we iterate through the n elements of A and update C, the value at index j of C corresponds to how many times j appeared in A. This step takes $O(n)$ time to iterate through A.

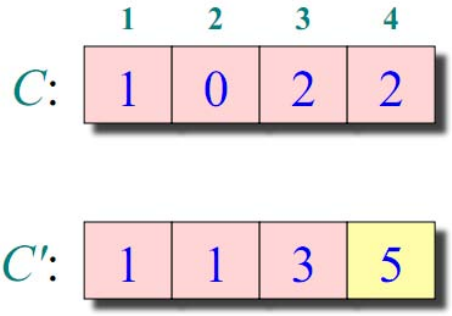


Once we have C, we can construct the sorted version of A by iterating through C and inserting each element j a total of $C[j]$ times into a new list (or A itself). Iterating through C takes $O(k)$ time.

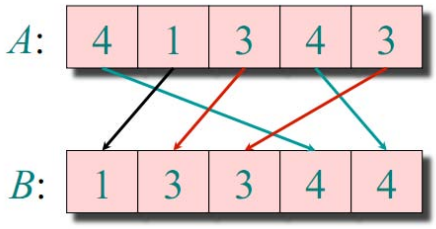
The end result is a sorted A and in total it took $O(n + k)$ time to do so.

However this does not permute the elements in A into a sorted list and is **not stable yet**. If A had two 3s for example, there's no distinction which 3 mapped to which 3 in the sorted result. We just counted two 3s and arbitrarily stuck two 3s in the sorted list. This is perfectly fine in many cases, but you'll see later on in radix sort why in some cases it is preferable to be able to provide a permutation that transforms A into a sorted version of itself.

Make it stable: We continue from the point where C is an array where $C[j]$ refers to how many times j appears in A. We transform C to an array where $C[j]$ refers to how many elements are $\leq j$. We do this by iterating through C and adding the value at the previous index to the value at the current index, since the number of elements $\leq j$ is equal to the number of elements $\leq j - 1$ (i.e. the value at the previous index) plus the number of elements = j (i.e. the value at the current index). The final result is an array C where the value of $C[j]$ is the number of elements $\leq j$ in A.



Now we iterate through A backwards starting from the last element of A. For each element i we see, we check $C[i]$ to find out how many elements are there $\leq i$. From this information, we know exactly where we can put i in the sorted array. Once we insert i into the sorted array, we decrement $C[i]$ so that if we see a duplicate element, we know that we have to insert it right before the previous i. Once we finish iterating through A, we will get a sorted list as before. This time, we provided a mapping from each element A to the sorted list. Note that since we iterated through A backwards and decrement $C[i]$ every time we see i. we preserve the order of duplicates in A. That is, if there are two 3s in A, we map the first 3 to an index before the second 3. This now makes counting sort **stable**. We will need the stability of counting sort when we use radix sort.



Iterating through C to change $C[j]$ from being the number of times j is found in A to being the number of times an element $\leq j$ is found in A takes $O(k)$ time. Iterating through A to map the elements of A to the sorted list takes $O(n)$ time. Since filling up C to begin with also took $O(n)$ time, the total runtime of this stable version of counting sort is $O(n + k + n) = O(2n + k) = O(n + k)$.

Radix Sort

Example

```
2341
1432
2413
1243
2143
```

```
1234
1342
2314
1423
2431
1324
2134
```

Is Heap Sort Stable?

No. An example of heap sorting $\{2, 1, 2\}$ can illustrate the point.

Is Merge Sort Stable?

Merge sort can be stable as long as the merge operaion is implemented properly.

Is the merge sort in `docdist6` stable?

```
1  def merge_sort(A):
2    n = len(A)
3    if n==1:
4      return A
5    mid = n//2
6    L = merge_sort(A[:mid])
7    R = merge_sort(A[mid:])
8    return merge(L,R)
9
10 def merge(L,R):
11   i = 0
12   j = 0
13   answer = []
14   while i<len(L) and j<len(R):
15     if L[i]<R[j]:
16       answer.append(L[i])
17       i += 1
18     else:
19       answer.append(R[j])
20       j += 1
21   if i<len(L):
22     answer.extend(L[i:])
23   if j<len(R):
24     answer.extend(R[j:])
25   return answer
```

No, due to the comparison at line 15. If two elements are equal, the element on the right will be put first in the merged array which changes the original ordering. If we change it to `L[i]`$\leq$`R[j]`, it will be stable.

6.006 Introduction to Algorithms
Fall 2011