Homework 3 Question 2, Solution by Cyrus

Estimate the actual worst-case time and space complexity of stable counting sort methods we discussed in class. We saw the standard reverse counting sort, then my similar forward version, and the linked list idea. Try to keep track of data movement (reading, copying, writing, deleting), constructing data structures (arrays, linked lists, pointers), comparison, usage of “if” statements, etc. You are permitted to optimize and use any tricks and implementations that you like, as long as the algorithms remain recognizable and correct.

Note: The following analysis is merely an example that, for instance, neglects to mention loop overhead, due to extremely complicated possible optimizations that could be made. Any solution that comes up with a consistent model of analysis that does not make unreasonable assumptions will receive full marks.

In the following analysis, it is assumed that pointers and values occupy the same amount of memory (1 unit), loops do not have associated overhead, and memory costs are linear (that is to say, a block of size \( k \) requires exactly \( n \) units of space).

Standard Reverse Counting Sort:
(assuming a nonempty input array)

We must first create an array of size \( k \), and while we’re at it, we might as well create an output array of size \( n \) now. We name the \( k \) array \( \text{temp} \) and the output array \( \text{out} \). This requires exactly \( n + k \) units of space, assuming that each space unit can hold values from \([0, \max(n, k)]\). We may either assume zeroed memory or incur a cost of \( n + k \) time here to zero it. If we consider the output array to already exist due to the nature of the question, then these costs become just \( k \).

The first phase of counting sort, the counting phase, must walk through the input array, and perform exactly \( n \) indexes into \( \text{temp} \), which we could consider its own operation, or we could consider an addition (to the array pointer). We also must perform \( n \) incrementations, which we could describe as a read, an addition, and a write.

The next phase creates the secondary size \( k \) array, where we store the number of values less than or equal to the index. This can actually be done in place over the original size \( k \) array allocated above, which is a pretty good optimization because it reduces the size cost by \( k \). Using this variant, we save ourselves a single write operation as well, because we can leave element \( \text{temp}_{c} \) as is. The pseudocode to perform this transformation is as follows:

for each \( i \in [1, k) \):
\[
\text{temp}_{i} \leftarrow \text{temp}_{i-1} + \text{temp}_{i}
\]

This loop accumulates a total of \( k - 1 \) additions, \( 2k - 2 \) reads, and \( k - 1 \) writes. If we are allowing the use of a finite number of registers, which function like memory sans read/write costs, then the reads in the previous statement are halved, because we can store the sum in a register in order to avoid writing it out and immediately reading it back in every iteration of the loop.

The final phase is a single reverse walk through the input array, which for each element, requires a read in the input, an index (pointer addition) into \( \text{temp} \), a decrement, and an index and write into \( \text{out} \).

The total cost is thus:

- \( n + k \) or \( k \) units of space
- \( 2n \) increments/decrements
- \( 2n + k - 1 \) additions
- \( 3n + k - 1 \) reads
- \( n + k - 1 \) writes

Two loops of size \( n \) and one loop of size \( k - 1 \) also occur, which may incur additional costs, depending on how you decide to analyze such control structures.

Forward Variant

Up to calculating the “secondary” \( \text{temp} \) array (\( B \) in the notes), the algorithm is the same as the above. The secondary array can be calculated in the same in place manner with minimal extra effort:

\[
c \leftarrow 0
\]
for each \( i \in [0, k) \):
\[
\text{temp}_{i} \leftarrow \text{temp}_{i}
\]
if \( (\text{temp}_{i} = 0) \):
\[
\text{temp}_{i} \leftarrow /
\]
else:
\[
\text{temp}_{i} \leftarrow c + 1
\]
\[
c \rightarrow c + v
\]

With the exception of \( k \) additional conditionals and comparisons, the analysis is pretty similar: \( k \) reads, \( 2k \) additions (the extra comes from adding the 1), and \( k \) writes. However, as mentioned in the notes, cells with value 0 (/ cells) are never used anyway, so we can leave the if statement and the first block out, and replace it with the second block, to remove the \( k \) conditionals and comparisons. Also, \( k \) of the additions (the \( k \) used to add 1) can be replaced with a single addition of 1 to a pointer representing the start of the output array (this manifests itself as a single extra addition).

The remainder of the analysis is pretty much the same; increments are replaced by decrements and the sweep is forward rather than backward. If we make all of these optimizations, the total costs come out to:
• $n + k$ or $k$ units of space
• $2n$ increments/decrements
• $2n + k + 1$ additions
• $3n + k$ reads
• $n + k$ writes

These results are extremely similar to that of the reverse counting sort: it is entirely possible that a bit of logic in the pseudocode loops could be modified to get them to have the same costs.

Linked List Variant:
The linked list idea would require an array of $2k$ pointers to reference the head and tail of each list (we would only need $k$ for the $\theta(n^2)$ worst case variant that does not keep track of the tail), $n$ (item, pointer) tuples to hold each item in the list representation, and a size $n$ array to hold the result. If we assume that pointers and values (and whatever data they may have associated with them) are the same size, we thus get a size cost of $2k + n + 2n$ units. However, unlike the previous 2 variants, this variant of counting sort holds all the data in the intermediate representation, and thus can overwrite the input array, for only $2k + 2n$ units.

When creating the initial arrays, we have some choices for the intermediate array of lists. We could initialize each (head, tail) pair to some flag stating that the list was empty, so we could set the head when the first item of the cell’s value is added, but this would require checking the flag for every add, for $n$ conditions on comparisons. We can do without by preallocating the first node of each linked list, and having an extra allocated cell at the end of each list, but this would incur a cost of $k$ extra linked list cells, for $2k$ extra memory units. I think the best solution is to instead initialize all cells to have the head and tail point to the cell itself, and upon a the first add, if we lay out the memory correctly, the head is updated implicitly (by the add operation) to point to the newly added cell. We then update the tail as well (as we must for every add). As if by magic, the same operation that can append to an existing list can convert an empty list to a single element list.

Thus to create the intermediate list representation, we iterate over the input array, and for each item, we:

1. Index into the list array.
2. For each item in input array, allocate a new linked list cell and copy the current input item’s value to the new cell. (1 copy, 1 alloc of size 2)
3. Dereference the tail pointer (which may point back to the cell containing the head/tail pointer).
4. Update the “next” (or implicitly “head”) item of the dereferenced cell to point to the newly allocated cell. (1 write)

Now, all that remains is to convert the linked list array to the output array, which we mentioned can be done over the input array. To do this, we must iterate over the array of linked lists, which requires $k$ reads for the end of each list, as well as $n$ incrememtaitions, and over the course of that entire operation, we require $n$ conditionals to check if the current item is the end of a list, $n$ dereferences, and $n$ copies, and finally $n$ incrementations of the output array index. The total cost of all operations comes out to:

• $n$ allocations (of size 2)
• $n$ copies
• $4n + k$ reads
• $n$ array indexes
• $2n$ writes

As we can see, we do a lot more work in this variant. All of the indirection and associated dereferencing manifests itself as a much higher number of reads than the other variants analyzed, and the higher memory cost is significant as well.

One might also note that in a practical sense, this algorithm is also quite poor, as the linear memory cost hypothesis is quite unrealistic, and a cache-aware analysis would certainly highlight more of the inadequacies of this technique.

Final note: This linked list can be simplified, and both the memory and time costs can be reduced using the following technique: Instead of using a tail pointer and appending each to the end of the linked list, prepend to the beginning of the list. This eliminates the need for tail pointers, and simplifies the logic off adding to the list (if there’s no tail pointer, there’s no need to update the tail pointer). The problem with this solution is that equal values have their order flipped\(^2\) ([1 3 4 4 4] $\rightarrow$ [1 3 4 4 4]). We could easily reverse each linked before outputting the final sorted array, but this issue is more efficiently remedied by doing the traversal of the original list backward. When this is done, the last element of a certain value is pushed to the list first, but it is displaced and pushed down by subsequent values. This solution requires only $2n + k$ additional space, and saves at least $n$ writes, depending on implementation.

---

\(^1\)By this I mean that the “head” value of the cells in the array of lists and the “next” value in a list cell need have the same offset, so we can treat a cell in the array like a list cell and have the head be updated to the “next” value.

\(^2\)We note that the original solution produces correctly ordered linked lists that may simply be joined together, and this solution produces lists that are reversed.