COMP2421

Sorting Algorithms Report

Computer Science Department

Mona Dweikat - 1200277

Dr. Radi Jarrar

23/2/2023

Table of Content

1. Cycle Sort

1.1: Description

Cycle sort is a comparison, unstable, in place sorting algorithm. The algorithm works to sort an array by starting at the first index and placing it at the correct index. The correct index is found by comparing the current element to all other elements, and counting the number of elements less than the current element, then adding the current index to that count. The algorithm goes through all elements of an array and cycles each element to its correct index. Each element in the array is considered to be the start of a cycle, because each element should be compared with every element of the array to be cycled to its correct index [1].

1.2: Algorithm

1. Start with the first index of the array and consider it to be the start of a cycle.
2. Count the number of elements less than the current element, notated n.
3. Find the correct index for the current element by adding n to the current index i.
4. Swap the current element to its correct index.
5. The start of the next cycle is the swapped element, and the program iterates over the entire array until it is sorted.

1.3: Time-Complexity

Table 1.3.1

Case	**Time Complexity**
Worst Case	O(n^2)
Average Case	O(n^2)
Best Case	O(n)

The worst case of cycle sort would be an array sorted in descending order, leading to a slow sorting algorithm, the number of cycles needed to sort the array correctly increases, since each element has to be compared to all other elements and be swapped to its correct index.

The average case time complexity depends on how the data is sorted, the sorting process could take O(n^2) or better.

The best case scenario would be if the data is sorted in an ascending order, it would take O(n) since there would be no swapping of elements, and only the comparison process would occur.

1.4: Space Complexity

The space complexity for the cycle sort algorithm is O(1), meaning that it uses a constant amount of memory regardless of the size of input data. This property of cycle sort is an advantage, since sorting is not taking up extra space in memory and is only swapping indices.

1.5: Stability

This algorithm is unstable, if there are duplicate elements, the output may not preserve the order of these elements after sorting. If an application deems it necessary to maintain the relative order of data, then cycle sort would not be the best sorting algorithm to use.

1.6: Memory Allocation

Cycle sort is an in-place sorting algorithm, the process of data manipulation does not take up more memory space, since it is comparing and cycling elements in the already allocated memory.

1.7: Example

Index	0	1	2	3	4
Element	5	2	0	4	3

For i staring at index 0, the first element to compare is 5, to be swapped to index n + i,
n = number of elements less than current element
i = current index
Count the number of elements less than 5, in this case n is equal to 4, and i is at index 0, 4 + 0 is 4, meaning 5 should be in index 4.

Index	0	1	2	3	4
Element		2	0	4	5

Index 0 is now empty, and 5 is in place of 3, since 3 still needs to be indexed, the next element to compare would be 3.
The number of elements less than 3 is 2, and i is at index 0, so 3 should be placed at index 2 + 0 = 2. 3 is now in place of 0.

Index	0	1	2	3	4
Element		2	3	4	5

Since index 0 is still empty, the first cycle is still not complete. 0 is the next element to be compared. The number of elements less than 0 are 0, and i is at index 0, so 0 should be indexed at 0 + 0 = 0.

Index	0	1	2	3	4
Element	0	2	3	4	5

The first index of the array [0] now has an element, so the first cycle is done. The algorithm goes through all elements until it has traversed the entire array and is sorted. In this case, the array was sorted using one cycle.

2. Cocktail Sort

2.1: Description

Cocktail sort is a variation of bubble sort, except it is bidirectional. In bubble sort, the array is iterated over from the starting index until the last index, and the items are swapped depending on the comparison. As for cocktail sorting, the array is iterated over twice, once from the first index to last, and one from the second to last index to the first index. After the first iteration, the largest element in the array is sorted to the end of the array, which is why the second iteration starts from the second to last element. This iteration process is repeated until the data is sorted. For the algorithm to know when to stop, there should be no swapping of elements, meaning the data was correctly sorted [2].

2.2: Algorithm

1. The first iteration starts with the first index i, and compares it with the second index j, if i is greater than j, the two items will be swapped.
2. i and j are incremented and compared again, until j is at the last index.
3. Since the last index j is now the largest element in the array, j and i are decremented.
4. j is now at the second to last element and i is at the element before, the two current elements are compared and swapped if needed.
5. The second iteration goes from the second to last element of the array until the first element.
6. This iteration process is repeated until no swapping of elements occurs, meaning the data is sorted.

2.3: Time-Complexity

Table 2.3.1

Case	Time Complexity
Worst Case	O(n^2)
Average Case	O(n^2)
Best Case	O(n)

The worst case of cocktail sort would be an array sorted in descending order, leading to a slow sorting algorithm, the number of iterations needed would increase if the data is reverse sorted.

The average case time complexity depends on how the data is sorted, the sorting process could take O(n^2) or better.

The best case scenario would be if the data is sorted in an ascending order, it would take O(n) since there would be no swapping of elements, and only the comparison process would occur.

2.4: Space Complexity

The space complexity for the cocktail sort algorithm is O(1), meaning that it uses a constant amount of memory regardless of the size of input data. Sorting is not taking up extra space in memory, since it is only swapping elements and the use of extra memory is not needed.

2.5: Stability

Since cocktail sorting is based on bubble sort, both algorithms are stable. If there are duplicate elements in an array, their relative order is maintained.

2.6: Memory Allocation

Cocktail sort is an in-place sorting algorithm, since it uses a constant amount of memory space.

2.7: Example

Index	0	1	2	3	4
Element	5	2	0	4	3
i/j iteration	i	j			

The first iteration starts at i = 0, and j = i + 1, 5 and 2 are compared and swapped.

Index	0	1	2	3	4
Element	2	5	0	4	3
i/j iteration		i	j		

i and j are iterated, 5 and 0 are compared and swapped.

Index	0	1	2	3	4
Element	2	0	5	4	3
i/j iteration			i	j	

After swapping 0 and 5, i and j are iterated, 5 and 4 are compared and swapped.

Index	0	1	2	3	4
Element	2	0	4	5	3
i/j iteration				i	j

j and i are iterated, and 5 and 3 are compared and swapped.

Index	0	1	2	3	4
Element	2	0	4	3	5
i/j iteration				i	j

The first iteration over the array is done, and the largest element in the array is at the last index.

Index	0	1	2	3	4
Element	2	0	4	3	5
i/j iteration			i	j	

In the second iteration, i and j are decremented, and 4 and 3 are compared and swapped.

Index	0	1	2	3	4
Element	2	0	3	4	5
i/j iteration		i	j		

i and j are decremented again, and 0 and 3 are compared, but not swapped since 3 is greater than 0.

Index	0	1	2	3	4
Element	2	0	3	4	5
i/j iteration	i	j			

After i and j are decremented, 2 and 0 are compared and swapped.

Index	0	1	2	3	4
Element	0	2	3	4	5
i/j iteration	i	j			

Since i and j are now back to the beginning of the array.
The algorithm iterates over the array the same way, and in this case since no swaps would occur, then the data is sorted and the iteration process ends.

3. Strand Sort

3.1: Description

Strand sorting algorithm is a recursive sorting algorithm that works on data structures that can easily be concatenated such as linked lists. The algorithm works by using a temporary sublist to sort the input data into an output list. The first element of the list is moved to the temporary sublist, and each following element is compared to the last element in the sublist, if it is greater, it is moved to the sublist, if it is less, it stays in the main list to be moved in later recursions [3].

3.2: Algorithm

1. A temporary sublist is created, and the first element of the input list is moved to the temporary list.
2. The node pointer p is incremented and checks if the element is less than or greater than the last element of the sublist, if it is greater, the element is moved to the sublist. If it is less, the element stays in the input list until the next recursion.
3. The algorithm goes over the list until pointer p is equal to NULL.
4. The resulting sublist of the first recursion is merged with the output list.
5. The algorithm goes over the list again, and the first element is moved to the sublist.
6. The list is traversed, and the algorithm is recursively executed until the output list is sorted.

3.3: Time-Complexity

Table 3.3.1

Case	Time Complexity
Worst Case	$O(n^2)$
Average Case	O(n log n)
Best Case	O(n)

The worst case of strand sort would be a list sorted in descending order, leading to a slow sorting algorithm, the number of recursions needed to sort the list would lead to a higher execution time.

The average case time complexity depends on how the data is sorted, the sorting process could take O(n log n), if the data is randomly or partially sorted.

The best case scenario would be if the data is sorted in an ascending order, it would take O(n). Since there would be only one element in the sublist, the first element, there would be no more than one concatenation of lists, leading to a faster execution time.

3.4: Space Complexity

The space complexity for the strand sort algorithm is O(n), meaning that it uses extra memory space. The algorithm uses extra memory space than allocated for the input list. Extra memory space is needed for the temporary sublist and for the output list.

3.5: Stability

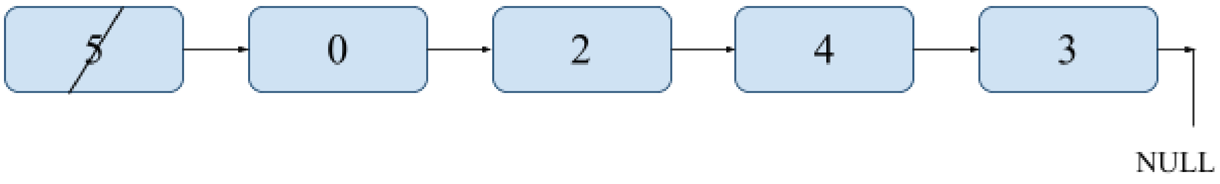
The strand sort algorithm is a stable sorting algorithm. If order of sorted duplicates is important, then this algorithm is a possible option.

3.6: Memory Allocation

Strand sort is not an in-place sorting algorithm. Since the output list and sublist use extra memory over the input array. The algorithm uses more memory to sort the list than the allocated memory for the input array.

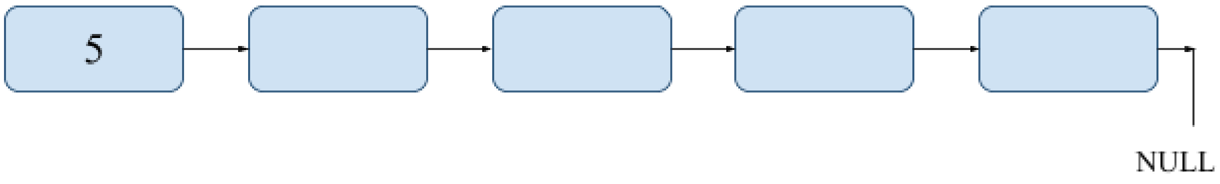
3.7: Example

Input List:



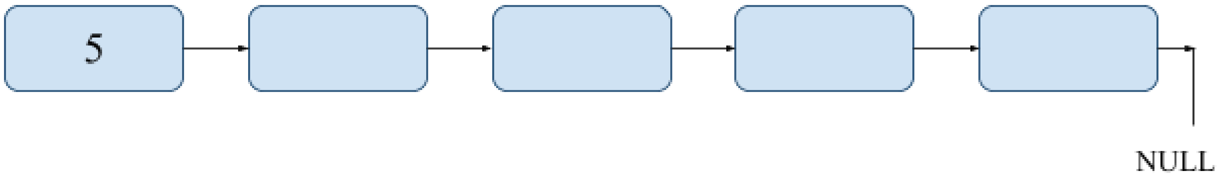
The input list is a randomly sorted list, the first element, 5 will be moved to a sublist.

Temporary Sublist:



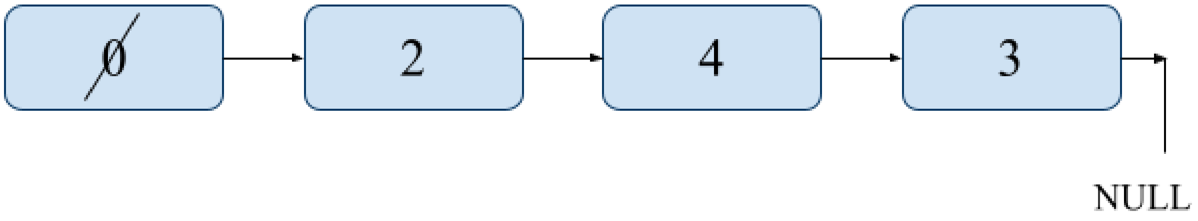
The list is then traversed to compare each element to 5. Starting at 0, 0 < 5, therefore 0 stays in the input list until the next recursion. Same goes for every element. Since no element is greater than 5, only 5 will end up in the sublist, and then be merged with the output list.

Output List:



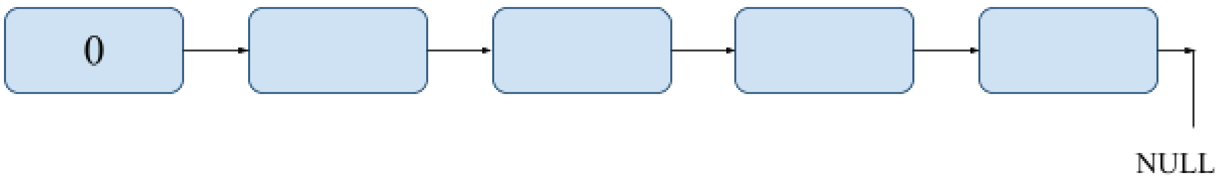
After merging the temporary list with the output list, a recursive call is executed to sort the rest.

Input List:



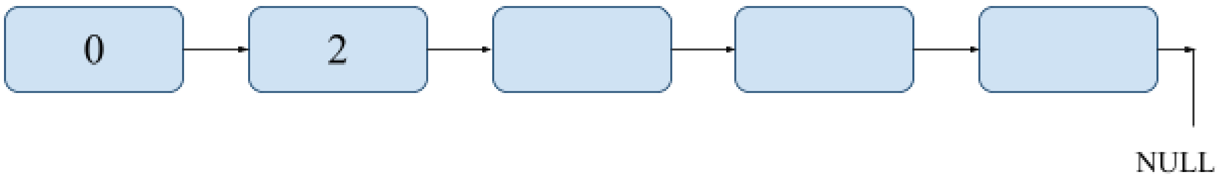
The first element 0, is moved to the temporary list.

Temporary Sublist:



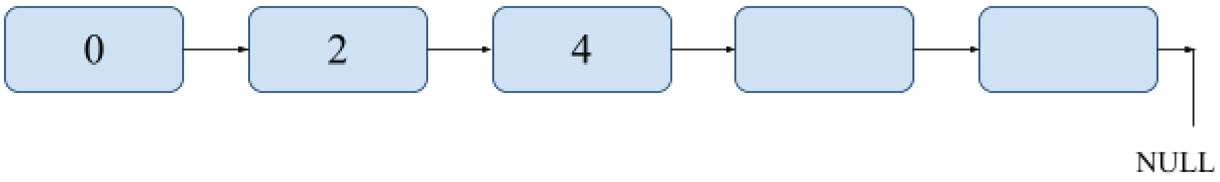
The list is traversed, and every element that is greater than 0 is inserted into the temporary list.

Temporary Sublist:



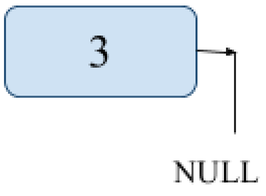
Since 2 is greater than 0, 2 is added to the sublist. 4 is then compared to the last element in the sublist, which is 2. 4 is greater than 2, so it is added to the list.

Temporary Sublist:

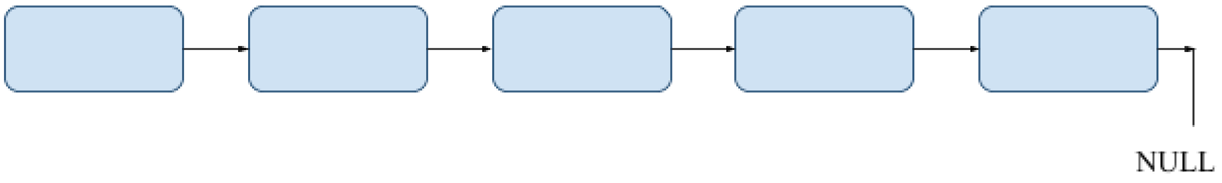


3 is now to be compared with 4, since 3 is less than 4, it stays in the input list.
Since traversal of the input list in the second recursion is over, the sublist and the output list are merged and sorted.

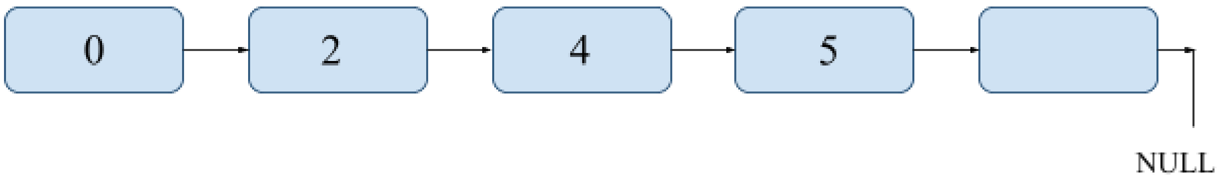
Input List:



Temporary Sublist:

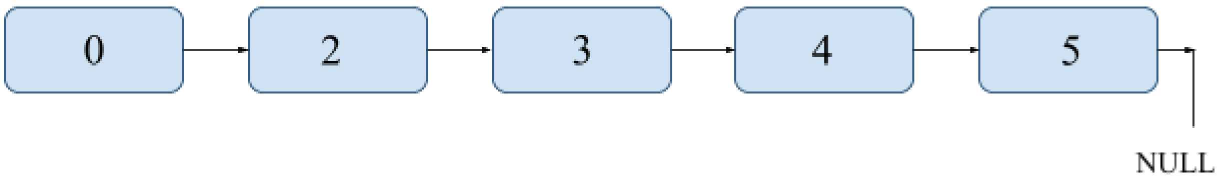


Output List:



The only element left is 3, so for the third recursion 3 will be the only element in the temporary sublist, and it will merge with the output list, giving the following list:

Output List:



The final output is a sorted linked list, the same algorithm could be applied to other data structures.

4. Bucket Sort

4.1: Description

Bucket sorting is an algorithm that divides an input into “buckets”, where each individual bucket is sorted using the same algorithm or a different one, such as insertion sort. These buckets could be any data structure that is flexible, such as a hash table, or an array of linked lists. After the individual buckets are sorted, they are then merged into a sorted output. This sorting algorithm is very efficient for sorting floating point numbers [4]. The array of linked lists is used to implement a hashtable, and the size of the array is determined by the load factor and the expected number of elements. The load factor depends on how full the table is expected to be, for example, a table is expected to be 75% full, the load factor would be 0.75. The number of buckets is then determined by dividing the expected number of elements over the load factor, and then rounding that number up to the closest integer (ceiling), this method is used for integers. For floating point numbers, they can be multiplied by a specific number (i.e 10) to convert them into integers, and then be hashed that way [5].

4.2: Algorithm

1. The algorithm determines the number of expected elements to be sorted.
2. The table size is determined using the load factor and the number of elements.
3. For floating point numbers, multiply by a large constant, and take floor of that number, this determines which index the number should be stored in.
4. The list is traversed, and each element is stored in its correct index in the hashtable, creating a linked list in each used index.
5. The individual buckets (lists) are sorted using any other sorting algorithm.
6. All sorted buckets are then merged together into an output list.

4.3: Time-Complexity

Table 4.3.1

Case	Time Complexity
Worst Case	O(n + k)
Average Case	O(n + k)
Best Case	O(n + k)

n: number of elements.

k: number of buckets.

The worst case of bucket sort would be an array sorted in descending order, the algorithm iterates over the full list, and be placed into their respective buckets. A reversely sorted array would not cause the algorithm to perform slower, since the iteration and bucket splitting is done regardless of how the input data is sorted.

The average and best case are also O(n + k), since the full process is done regardless of how the data is spread in the list. The average case being a randomly distributed data, and the best case being the data is sorted. The traversal and bucketting of the input is done regardless of the input.

4.4: Space Complexity

The space complexity for the bucket sort algorithm is O(n), meaning that it uses extra memory space. The algorithm uses extra memory space than allocated for the input list. Extra memory space is needed for the hashtable and the linked lists formed through the hashing process.

4.5: Stability

The bucket sort algorithm may or may not be a stable sorting algorithm. If buckets are sorted using a stable algorithm, such as insertion sort, then bucket sort is stable. If the buckets are sorted using an unstable algorithm, such as selection sort, then bucket sort is unstable.

4.6: Memory Allocation

Bucket sort memory allocation depends on the implementation. Bucket sorting is mostly not an in-place algorithm, since it uses extra memory space for the hashtable, linked lists, and output lists. Depending on the implementation, bucket sort could be in-place. Using an in-place bucket sorting algorithm, where buckets are created within the original array.

4.7: Example

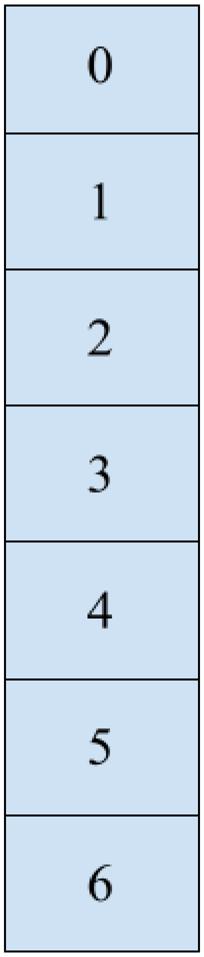
Index	0	1	2	3	4
Element	0.4	0.2	0.5	0.1	0.3
i iteration	i				

The number of elements in the array is 5 elements, and assuming extra elements could be added, and the table could be 75% full, the table size could be determined by dividing the expected number of elements (5) over the load factor (0.75), and taking its ceiling value.

Table size = 5 / 0.75 = ceil (6.67) = 7

i iterates over the table to convert the decimals to integers, then store them into their respective buckets.

Hashtable:



Iterating over the elements of the table to convert them to integers:

Integer = floor (element * 10)

Table 4.7.1

Index of Input	**element**	**Integer = floor (element * 10)**
0	0.4	4
1	0.2	2
2	0.5	5
3	0.1	1
4	0.3	3

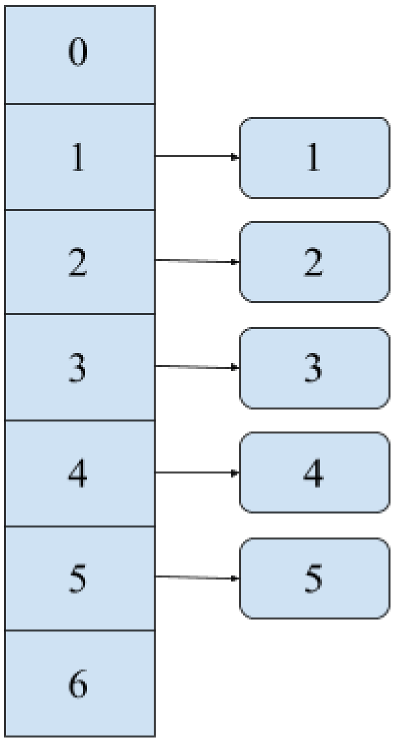
To find the respective index for each element, index = element % table size.

Table 4.7.2

Index of Input	Element	Index = element % table size
0	4	Index = 4 % 7 = 4
1	2	Index = 3 % 7 = 2
2	5	Index = 5 % 7 = 5
3	1	Index = 1 % 7 = 1
4	3	Index = 3 % 7 = 3

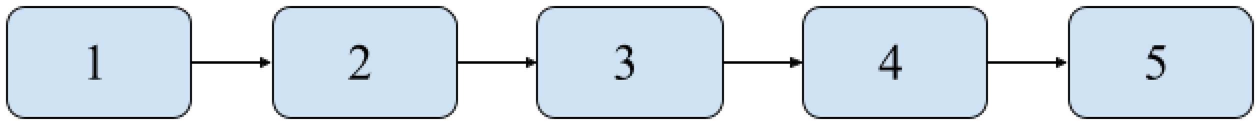
After determining where each element should be indexed into the hashtable, they are added to their respective index, and then sorted.

<u>Hashtable:</u>



Looking at each linked list at each index, all lists are sorted in an ascending order. All buclets in the hashtable are then merged into an output list.

Output List:



5. Comb Sort

5.1: Description

Comb sort is another algorithm that improves upon the bubble sort algorithm. In bubble sort, the algorithm uses a comparison gap of 1, as opposed to the comb sort algorithm, which uses a larger gap, and this gap decrements by a factor of 1.3, and iterates until the gap is at 1. The 1.3 shrink factor was determined after testing the comb sort algorithm over a very large number of random lists. After each shrink, the array is traversed using the gap size to compare and swap elements, these are called runs [6].

5.2: Algorithm

1. The algorithm determines the gap size using the size of the array, if an array is of size 5, the gap size would be 5.
2. For the first run, the first and last elements are compared and swapped if necessary.
3. The gap is shrunk by a factor of 1.3, and the elements are traversed, compared and swapped.
4. The gap shrinks until the value is 1, this signifies the last run. The array is once again traversed, each element is compared to its adjacent element and swapped if needed. This run completes the traversal of the array, and the sorting process is done.

5.3: Time-Complexity

Table 5.3.1

Case	**Time Complexity**
Worst Case	O(n^2)
Average Case	O(n^2)
Best Case	O(n log n)

The worst case of comb sort is O(n^2) if an array is reverse sorted, the algorithm would need to iterate over the list n number of times, and with each iteration swaps are bound to happen, leading to a slow algorithm. The gap factor would not do much improvement in such a case.

For the average case, with the data randomly dispersed, the algorithm still takes O(n^2). The iteration and swapping process still needs to happen, meaning the time complexity would not be improved in the average case.

The best case is improved to O(n log n). If the data is already sorted, the algorithm only traverses and compares the elements without swapping.

5.4: Space Complexity

The space complexity for the comb sort algorithm is O(1), meaning that it does not use extra memory space.The algorithm compares and swaps the elements within the allocated memory for the input array.

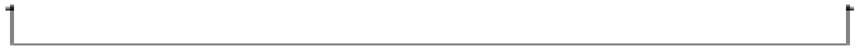
5.5: Stability

The comb sort algorithm is an unstable algorithm, since the relative order of duplicates is not maintained. The swapping of elements could cause mix ups in maintaining the order of duplicates.

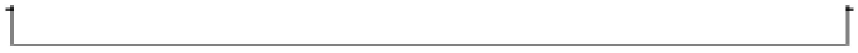
5.6: Memory Allocation

Comb sort is an in-place sorting algorithm, since it uses the exact allocated memory for the input array, without moving out of that allocated area for extra memory. The sorting is only done within the input array.

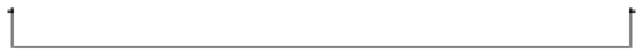
5.7: Example

Index	0	1	2	3	4
Element	5	2	0	4	3
gap					

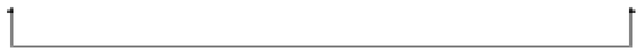
The gap starts at value 5, same as the array size. 5 and 3 are compared, since 3 is less than 5, they are swapped.

Index	0	1	2	3	4
Element	3	2	0	4	5
gap					

To find the value of the next gap size, the previous gap size is shrunk by a factor of 1.3. In this case the next gap size is 5 / 1.3 = 3

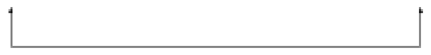
Index	0	1	2	3	4
Element	3	2	0	4	5
gap					

The gap is shrunk to a value of 3, and 3 and 4 are compared. Since 4 is greater than 3, no swapping occurs and the gap is incremented.

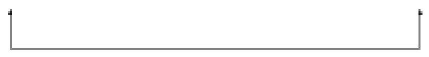
Index	0	1	2	3	4
Element	3	2	0	4	5
gap					

The elements 2 and 5 are compared but not swapped because 5 is greater than 2. Since the gap is at the last element, it is shrunk again.

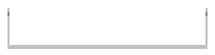
3 / 1.3 = 2

Index	0	1	2	3	4
Element	3	2	0	4	5
gap					

Elements 0 and 3 are compared and swapped. The gap is incremented to compare the next values.

Index	0	1	2	3	4
Element	0	2	3	4	5
gap					

After incrementing the gap, 2 and 4 are compared, but not swapped. The gap increments again until the end of the array. No swaps occur. The gap is shrunk again to 2 / 1.3 = 1.

Index	0	1	2	3	4
Element	0	2	3	4	5
gap					

The gap now compares adjacent elements and swaps if necessary. The gap is incremented to go over the entire array. After the gap reaches the end of the array without any swaps, the array is sorted.

Summary

Table 6.1

Algorithm	Time Complexity	Space Complexity	Stability	Memory Allocation
Cycle Sort	O(n^2)	O(1)	Unstable	In-Place
Cocktail Sort	O(n^2)	O(1)	Stable	In-Place
Strand Sort	O(n^2)	O(n)	Stable	Not in-place
Bucket Sort	O(n + k)	O(n)	Either	Either
Comb Sort	O(n^2)	O(1)	Unstable	In-Place

In conclusion, there is no "best" sorting algorithm to use, since the choice of algorithm depends on the goal of the program. If a program aims for speed, then bucket sorting is the best choice, but if the program needs to be contained in a restricted memory area, then bucket sorting wouldn't be the ideal choice. A better choice would be cycle, cocktail, or comb sorting. If the program data needs to be sorted with respect to the order, cocktail sorting or strand sorting is the way to go.

The choice of sorting algorithm also depends on the data itself, the type of the given data. For integers, any of the above algorithms are efficient. For floating point numbers, bucket sorting would be the most effective, because of how floating point numbers are distributed, making it efficient to divide them into buckets and sorting each bucket individually. Bucket sorting would also be very efficient to sort strings, the input strings can be hashed to integers, and these integers be sorted in the buckets.

Another very important thing to keep in mind when choosing a sorting algorithm is the size of the input data. If the data size is small, cycle, strand, and cocktail sorting would be most efficient in dealing with less data. Strand sorting would be very inefficient for larger data since it uses too much space and the number of recursions would increase with more data. Considering these three algorithms, cocktail sort would be better than strand and cycle sorting. Since cocktail sorting is in-place and has a space complexity of O(1), and is stable. Bucket and comb sorting would be very efficient with larger data sizes. Since comb sorting is an enhancement of bubble sorting, it can work very well with large data since it can decrease the number of element swapping with the larger gap. Bucket sorting is especially efficient for large data that is uniformly distributed. The uniform distribution minimizes the number of collisions in the hashtable. For larger data sizes, bucket sorting would be more effective if memory allocation isn't a restricting factor, but if memory is limited, then an in-place sorting algorithm would work better, such as comb sorting.

References

1) “Cycle Sort.” *GeeksforGeeks*, GeeksforGeeks, 15 Feb. 2023, www.geeksforgeeks.org/cycle-sort/. (Accessed: 14th February 2023)
2) “Cocktail Sort.” *GeeksforGeeks*, GeeksforGeeks, 19 July 2022, www.geeksforgeeks.org/cocktail-sort/. (Accessed: 15th February 2023)
3) *YouTube*, YouTube, 25 Apr. 2021, www.youtube.com/watch?v=jHzFJU2-oVA. (Accessed 22 Feb. 2023)
4) Heineman, G. T. (n.d.). *Algorithms in a Nutshell*. O'Reilly.
5) “Load Factor and Rehashing.” *GeeksforGeeks*, GeeksforGeeks, 21 Jan. 2023, www.geeksforgeeks.org/load-factor-and-rehashing/. (Accessed: 23rd February 2023)
6) “Comb Sort.” *GeeksforGeeks*, GeeksforGeeks, 10 Jan. 2023, www.geeksforgeeks.org/comb-sort/. (Accessed 23 Feb. 2023)