*Title and Author:*

Time and Space Complexity of Sorting algorithms

*By Ken Stearns*

*Abstract:*

Sorting algorithms come in many shapes and sizes. Most of these have specific use cases that allow them to shine in their best possible light. But there are cases where what sorting algorithm to choose isn’t so obvious. In the computer science world, there are many different factors that will help one decide on how to sort their data. Two of the larger factors that we are going to focus on are the time and space complexity of each algorithm, with no concerns about how the algorithm works. We will be looking at the following algorithms:

* Insertion Sort
* Bubble Sort
* Merge Sort
* Heap Sort
* Radix Sort
* Count Sort
* Selection Sort
* TimSort
* Pigeonhole Sort
* Bitronic Sort
* Comb Sort

Using these algorithms as a base line will help get an understanding of how to see the other algorithms that exist.

*Introduction:*

The ability to organize dirty data into usable data is a very sought-after skill in development. When going through schooling some of the earlier subjects that we first encounter are some basic sorting algorithms and learning the pros and cons of a select few. As a current senior in college, I would like to pursue more of these sorting algorithms to broaden my experience with them. Currently I have been learning programming for about 6 years with one summer internship working on front end development. Sorting “dirty” data into usable data is used widely across all fields of computer science. While what will need to be sorted will differ greatly from specialty to specialty, the constant will always be the need to clean up data.

*Literature review:*

*Body:*

For the entirety of this experiment, we will create one string of numbers that is created randomly that will be used for every type of sorting algorithm to minimize outlying variables. The text file that was used for the source of numbers was generated using C++ and contains 10000 different numbers. Going into each type of sorting algorithm, we will need a base understanding of how they work to properly test each of them. The following is a description of each type of sorting algorithm.

Insertion Sort is one of the more basic sorting algorithms. This algorithm works by involving the “sorted list created based on an iterative comparison of each element in the list with its adjacent element” (*Nvidia*) This works by utilizing nested loops, one that increases the index for the original list and the other to compare items adjacent to the index. When the item that is being compared to is larger, the second loop will move the compared item one to the right and will continually do this until the compared item is not larger. Then the loop will insert the indexed item at that location. Time wise this sorting method utilizes two nested loops, which results in an O(n^2). Space-wise this algorithm does not create any new arrays and just requires one more integer to be declared, which gives it O(1). After running the test program, I have gathered the following results. The total number of operations is 25325880 and the average time in microseconds was 40522. Here’s the raw data for the results from the test:

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 25335879 | 39799 |
| 2 | 25335879 | 40196 |
| 3 | 25335879 | 40090 |
| 4 | 25335879 | 43754 |
| 5 | 25335879 | 40149 |
| 6 | 25335879 | 40668 |
| 7 | 25335879 | 40154 |
| 8 | 25335879 | 39946 |
| 9 | 25335879 | 39809 |
| 10 | 25335879 | 40653 |

Bubble Sort is very similar to the insertion sort algorithm but goes in reverse order. “Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order” (*Programiz*) Much like the insertion sort this algorithm utilizes two loops. This algorithm will compare the two elements directly adjacent to each other, and if the element on the left is larger than the element on the right they will be swapped. Then the algorithm will proceed to the next two elements. Once we reach the end of the list, the algorithm will go back to the start of the list and go through the list again repeatedly until no swaps are needed. The time complexity of this sorting boils down to having two nested loops, which results in O(n^2). Space complexity of this also does not require any new arrays or anything created resulting in O(1). After running the test program, here are the results for the bubble sort. The sorting was finished after 99950004 operations in an average of 222579 microseconds. Here’s the raw data for the results:

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 99950004 | 221532 |
| 2 | 99950004 | 220289 |
| 3 | 99950004 | 221853 |
| 4 | 99950004 | 223336 |
| 5 | 99950004 | 222268 |
| 6 | 99950004 | 223297 |
| 7 | 99950004 | 228575 |
| 8 | 99950004 | 221650 |
| 9 | 99950004 | 220244 |
| 10 | 99950004 | 222748 |

Merge Sort is widely known as one of the more efficient sorting algorithms. “It works on the principle of Divide and Conquer based on the idea of breaking down a list into several sub-lists until each sub-list consists of a single element and merging those sub-lists in a manner that results into a sorted list.” (*DigitalOcean*) Space complexity of a merge sort is very ineffective as it requires two more arrays to be created, giving it O(n). Time complexity will result in a O(nlgn) Merge sort resulted in a total of 133616 operations and an average time of 1603 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 133616 | 2002 |
| 2 | 133616 | 2002 |
| 3 | 133616 | 1501 |
| 4 | 133616 | 1501 |
| 5 | 133616 | 2005 |
| 6 | 133616 | 1501 |
| 7 | 133616 | 1509 |
| 8 | 133616 | 1501 |
| 9 | 133616 | 1501 |
| 10 | 133616 | 1002 |

Heap Sort is the next algorithm we will tackle. “Heapsort is similar to selection sort – we’re repeatedly choosing the largest item and moving it to the end of our array. The main difference is that instead of scanning through the entire array to find the largest item, we convert the array into a max heap to speed things up.” (*Interview Cake*) Space complexity for Heap sort results in O(1), and the time complexity results in O(logn) Heap sort resulted in 290241 operations and an average time of 1527 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 290241 | 1501 |
| 2 | 290241 | 1501 |
| 3 | 290241 | 1502 |
| 4 | 290241 | 1501 |
| 5 | 290241 | 1500 |
| 6 | 290241 | 1502 |
| 7 | 290241 | 2001 |
| 8 | 290241 | 1000 |
| 9 | 290241 | 1764 |
| 10 | 290241 | 1501 |

Radix Sort is used mostly for lists of integers that contain more than one digit. “Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digits to the most significant digit.” (*Javatpoint*) Space complexity results in a O(n + k) where k is the number of digits, and time complexity results in O(n \* k). Radix sort resulted in 160049 operations in an average time of 1255 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 160049 | 500 |
| 2 | 160049 | 1510 |
| 3 | 160049 | 1502 |
| 4 | 160049 | 1001 |
| 5 | 160049 | 1001 |
| 6 | 160049 | 1518 |
| 7 | 160049 | 1501 |
| 8 | 160049 | 1001 |
| 9 | 160049 | 2011 |
| 10 | 160049 | 1001 |

Selection Sort is one of the simplest sorting algorithms out there. “A selection-based sorting algorithm is described as an in-place comparison-based algorithm that divides the list into two parts, the sorted part on the left and the unsorted part on the right. Initially, the sorted section is empty, and the unsorted section contains the entire list.” (*Simplilearn*) Time complexity for selection sort results in O(n^2) and the space complexity results in O(1). Selection sort came in with 50004999 operations and an average time of 61409 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 50004999 | 60727 |
| 2 | 50004999 | 62173 |
| 3 | 50004999 | 60839 |
| 4 | 50004999 | 61109 |
| 5 | 50004999 | 61068 |
| 6 | 50004999 | 61144 |
| 7 | 50004999 | 60872 |
| 8 | 50004999 | 64676 |
| 9 | 50004999 | 60813 |
| 10 | 50004999 | 60673 |

TimSort is one of the newer sorting algorithms that is being looked at today. “The main Timsort algorithm consists of three phases – we calculate a run length, sort each run of elements using insertion sort, and then recursively sort adjacent runs using merge sort. (*Baeldung*) Space complexity wise TimSort results in a O(n), and time complexity at a O(nlogn) TimSort resulted in 260043 operations and an average time of 601 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 260043 | 501 |
| 2 | 260043 | 1004 |
| 3 | 260043 | 501 |
| 4 | 260043 | 501 |
| 5 | 260043 | 500 |
| 6 | 260043 | 500 |
| 7 | 260043 | 1001 |
| 8 | 260043 | 501 |
| 9 | 260043 | 501 |
| 10 | 260043 | 501 |

Pigeonhole Sort is one of the few sorting algorithms that does not compare elements. “Following the same principles of Pigeon Holes in mail services, in programming we can abstract three steps: Constructing a cabinet with enough number of pigeon holes, classifying the objects in its proper hole and recollect the object from the cabinet now in order.” (*Medium*) Time complexity results in O(n + range) and space complexity results in O(range), where the range is the range of the numbers being sorted. Pigeonhole sort resulted in 62762 operations in an average time of 2524 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 62762 | 3003 |
| 2 | 62762 | 2513 |
| 3 | 62762 | 2505 |
| 4 | 62762 | 2002 |
| 5 | 62762 | 2502 |
| 6 | 62762 | 2502 |
| 7 | 62762 | 2158 |
| 8 | 62762 | 2533 |
| 9 | 62762 | 2506 |
| 10 | 62762 | 3017 |

Bitronic Sort is one of the few sorting algorithms I have no experience with prior to this experiment. “Bitronic sort is a comparison-based sorting algorithm that can be run in parallel. It focuses on converting a random sequence of numbers into a bitronic sequence, onthat monotonically increases, then decreases. Rotations of bitonic sequence are also bitonic.” (*Rutgers*) Space complexity results in a O(n log^2 n) and time complexity results in a O(log^2 n). Bitronic sort resulted in 539982 operations and an average time of 3488 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 539982 | 4013 |
| 2 | 539982 | 3503 |
| 3 | 539982 | 3290 |
| 4 | 539982 | 3502 |
| 5 | 539982 | 3549 |
| 6 | 539982 | 3004 |
| 7 | 539982 | 3002 |
| 8 | 539982 | 3503 |
| 9 | 539982 | 4013 |
| 10 | 539982 | 3504 |

Comb Sort is the final algorithm that is being looked at. “Comb sort is a comparison-based sorting algorithm which improves on bubble sort… Comb sort uses a larger gap and works on bubble sort strategy.” (*iDeserve*) Time complexity results in a best case of O(n log n) and worst case of O(n^2), where space complexity results in O(1). Comb sort resulted in 316770 operations and had an average time of 1202 microseconds.

|  |  |  |
| --- | --- | --- |
| Run # | # of Operations | Time (microseconds) |
| 1 | 316770 | 1001 |
| 2 | 316770 | 1001 |
| 3 | 316770 | 1001 |
| 4 | 316770 | 1501 |
| 5 | 316770 | 1001 |
| 6 | 316770 | 2014 |
| 7 | 316770 | 1000 |
| 8 | 316770 | 1001 |
| 9 | 316770 | 1501 |
| 10 | 316770 | 1001 |

*Conclusions/Discussions:*

As we found out with this one test case, the results vary greatly across the 10 tested sorting algorithms. The sorting methods that were expected to perform faster also ended up taking up the most amount of space. While some of the results that were achieved here do not directly reflect the applicable time and space complexities, most of the time complexities posted above are worst-case scenarios. Moving forward with this project the next steps would be to test out more data sets along with on more varied machines. Some of the discrepancies above can likely be explained by background tasks happening on the pc that was running those tests. Also, another next step would be to track the specific RAM usage for each sorting method on top of tracking the time. Another potential issue that could come up is the sorting algorithms that require more operations will take O(n) more time to complete in this testing environment due to the process that was used to track the total number of operations. All in all, try out different sorting methods for your specific situation as all of these methods have pros and cons that come with them.

*Bibliography (Only contains URLs currently but will be flushed out):*

<https://developer.nvidia.com/blog/insertion-sort-explained-a-data-scientists-algorithm-guide/>

<https://www.programiz.com/dsa/bubble-sort>

<https://www.digitalocean.com/community/tutorials/merge-sort-algorithm-java-c-python>

<https://www.interviewcake.com/concept/java/heapsort>

<https://www.simplilearn.com/tutorials/data-structure-tutorial/selection-sort-algorithm#:~:text=Selection%20sort%20works%20by%20taking,of%20the%20array%20is%20unsorted>.

<https://www.javatpoint.com/radix-sort>

<https://www.baeldung.com/cs/timsort>

<https://medium.com/@saishaddai/pigeon-hole-sort-algorithm-b8bcb0d63bc7>

<https://people.cs.rutgers.edu/~venugopa/parallel_summer2012/bitonic_overview.html>

<https://www.ideserve.co.in/learn/comb-sort>