Andy Cherney

Professor Mark Brody

CS 502 - 001

17 February 2024

Gnome Sort: A Simple but Inefficient Way to Arrange Flowerpots

Introduction

Ever picture a garden gnome sorting flowerpots? How would he go about it? If these two questions sound odd to you, you are not alone. If I were to tell you that there is a whole algorithm dedicated to this premise, you would think call me crazy. However, the logic behind it is actually quite easy to understand. There are many sorting algorithms out there each with their own advantages and drawbacks. While simplicity is always good, it may not always mean that the algorithm would be the most optimal and efficient solution. Similarly, an algorithm can be complex but efficient. Gnome sort though, tends to be a strong example of the first case, which is why it should only be used for educational purposes or whenever the dataset is small (*Gnome Sort*).

This report will highlight the overall background of the algorithm as well as how it works. The implementation of this algorithm will be carried out in 2 languages: C and Python. Finally, this report will also compare various test cases and time results of both implementations to evaluate its performance on sorting arrays of integers.

History

There is not a whole lot of history to this algorithm and the true intentions behind its implementation are not so clear. What we do know is that it was first proposed in 2000 by Iranian Computer Science Professor, Hamid Sarbazi-Azad, at the Sharif University of Technology under the pseudonym “stupid sort”. Dutch computer scientist, Dick Grune, later renamed it to gnome sort after the standard Dutch Garden Gnome (Grune). Upon its publication, Azad had this to say

“In 2000, when I was a PhD student at Glasgow University, I published a note in the departmental newsletter (issue 599, 2 October 2000) on a simple sort algorithm that had only one loop and called it Stupid Sort (you surely know that sort algorithms are either recursive or non-recursive but with two nested loops). Several years later, the algorithm has been reported as Gnome sort by Dr Dick Grune. It seems that the algorithm is getting more importance!”(Azad)

Despite what the pseudonym “stupid sort” suggests, this algorithm is not to be confused with bogosort, which was also published under the same pseudonym and is nothing but a joke due to its way of sorting based on pure randomization (*Bogosort or Permutation Sort*). Rather, it is referred to in that way because if you were to imagine a garden gnome trying to sort your flowerpots by size, he takes a “too direct approach” to solve the problem (Xenopax). So, even if the algorithm is simple with one loop as Azad claims, it may not always be enough, which means there aren’t many practical use cases for it. The next section will elaborate on this point in further detail.

Pseudocode

The logic is simple. As the gnome walks down the array of elements (the flowerpots), he looks at the one next to him and compares it to the previous one. If the value (the size of the flowerpot) of the front is greater than or equal to the one behind him, he does nothing and moves up by 1 element in the array. Otherwise, he swaps the left and right values and moves back down by 1 element. If there is no previous element, he moves back up. The process ends once he reaches the end of the array (Grune).

Figure 1.1 is the pseudocode for the Gnome-Sort algorithm assuming an array index starting from 0.

```
function gnomeSort(A[]):
    position = 0
    while position < length(A) do
        if position at beginning or A[position] >= A[position - 1]
            then position = position + 1
        else
            swap A[position] and A[position - 1]
            position = position - 1
```

Figure 1.1: Gnome Sort Algorithm Pseudocode

Now, we let N represent the size of the array.

Although it is easy to assume that the time complexity of this algorithm is linear due to having only one loop running up to N, this would only hold true in the best-case scenario where the array is almost sorted. However, if we imagine the worst-case scenario where the array was reverse sorted, for every k steps the gnome takes to move up in the array, he must take a total of $2k$ steps just to return to his original position, since he must move down when swapping the elements every time. As a result, the worst-case time complexity is actually $O(N^2)$, matching algorithms like bubble and insertion sort. It actually operates like a hybrid between the insertion sort and bubble sort algorithms, having almost the same number of comparisons. The only difference is just the one loop (Xenopax). The constant pairwise comparison as well as moving back and forth is the reason why it is not very efficient and why Xenopax refers to it as a "too direct approach."

Implementation 1

Figure 2.1 is the C implementation of the Gnome sort algorithm.

```c
void gnomeSort(int* A, int size){
    int i = 0;
    while (i < size){
        if (i == 0 || A[i] >= A[i - 1]){
            i++;
            continue;
        }
        int temp = A[i];
        A[i] = A[i - 1];
        A[i - 1] = temp;
        i--;
    }
}
```

Figure 2.1: C Code for Gnomesort

In C, this algorithm is not that complex to implement and there really was no struggle at all. The code tends to match the pseudocode so easily that many of the operations are directly translatable line by line. I started by defining the index variable that represents the gnome, then implemented the loop conditions and swap operation. Apart from syntax, one thing that is different with this implementation is the extra size parameter. I wanted to manually pass in the size as to not affect the timing too much, since C does not have a way to calculate length of the array when the arrays are dynamically allocated and passed into functions. Additionally, a temp variable is needed just like in most languages, since there is no way to swap elements in place. Other differences like the continue keyword instead of an else statement are just stylistic choices. However, overall, this algorithm was quite easy to implement, and being able to translate algorithms to a given programming language from pseudocode is a good skill to have in the long run.

Implementation 2

The second implementation is done in Python. Figure 3.1 is the code for it.

```
def gnomeSort(A):
    i = 0
    while i < len(A):
        if (i == 0 or A[i] >= A[i - 1]):
            i += 1
            continue

        A[i], A[i - 1] = A[i - 1], A[i]
        i -= 1
```

Figure 3.1: Python Code for Gnome Sort

The Python implementation is even easier to implement because the syntax matches the pseudocode more. It even takes the same number of lines to translate directly. Therefore, I followed the same strategy of just looking at the pseudocode and translating it line by line. Python is a language that supports variable unpacking, so the swap operation can be consolidated into one line with no temp variables needed. Additionally, there is no need to pass in the size manually because Python already has a built-in length function. It also happens to run at constant time because lists in Python are dynamic by design. The length is a counter that automatically gets updated as the list grows, so it will not influence the runtime in any way (*Internal Working of the Len() Function*). Again, implementing this just provided another opportunity to translate pseudocode to a programming language and with Python, there is almost no thinking involved because of how closely the syntax matches to it.

Comparison

Overall, there is not a whole lot of difference between the pseudocode and the 2 implementations, which further demonstrates the simplicity of this algorithm. They both start by first looping over the array with a while loop and check if the next value of the array is larger than or equal to its previous value. If true, then it will do nothing and just increment the index. Otherwise, it will swap the 2 elements and move the index back by 1. Syntactically though, Python has more similarity to the pseudocode.

The biggest difference really is the swap operation. In Python, it is possible to swap 2 array elements in place using only one line of code. However, in C and many other languages, there is no such thing. Therefore, a temporary variable needs to be created to hold one of the elements before the swap takes place. 3 lines of code are needed to accomplish the whole operation. However, in either case, this has no effect on the runtime of the algorithm itself. Another difference is just a stylistic choice of using the continue keyword to move to the next iteration of the loop. This can be done since each statement executes in either the if block or the else block. Apart from these, any other differences are just small variations in syntax. Finally, the C implementation also has an extra size parameter since passing in the size of the array directly is faster than looping over each element to find its length first.

Testing

Testing was run on both implementations, but the logic for the tests was fully implemented in Python. To run the C implementation, the subprocess module was used to execute the program, pass in each test case to stdin, and capture the sorted output. The first value of each input was the array size and then each element was passed in line by line. For all tests except the first 2 edge cases, the arrays were randomly generated with the RNG having a lower bound of -50 and an upper bound of 100. This was done to demonstrate the algorithm handling both positive and negative values. Table 1 is a set of 20 tests to demonstrate the algorithm's effectiveness.

#	Size	Test case	C result	Python result	Passed
1	0	[]	[]	[]	True
2	1	[9]	[9]	[9]	True
3	14	[-13, -15, -47, 84, 2, -2, -47, 67, 5, -35, -8, 77, 97, 79]	[-47, -47, -35, -15, -13, -8, -2, 2, 5, 67, 77, 79, 84, 97]	[-47, -47, -35, -15, -13, -8, -2, 2, 5, 67, 77, 79, 84, 97]	True
4	15	[48, -3, 1, 20, 21, 2, -42, 97, 22, -2, -4, 96, 35, 1, 51]	[-42, -4, -3, -2, 1, 1, 2, 20, 21, 22, 35, 48, 51, 96, 97]	[-42, -4, -3, -2, 1, 1, 2, 20, 21, 22, 35, 48, 51, 96, 97]	True
5	13	[40, 26, 88, -25, -2, -30, 54, -10, 34, 87, 81, 29, 90]	[-30, -25, -10, -2, 26, 29, 34, 40, 54, 81, 87, 88, 90]	[-30, -25, -10, -2, 26, 29, 34, 40, 54, 81, 87, 88, 90]	True
6	12	[-24, -50, 62, 25, 51, 69, -45, 81, -6, 21, -14, 27]	[-50, -45, -24, -14, -6, 21, 25, 27, 51, 62, 69, 81]	[-50, -45, -24, -14, -6, 21, 25, 27, 51, 62, 69, 81]	True
7	12	[6, 21, 1, 71, 32, -10, 32, -12, 94, 18, -20, -1]	[-20, -12, -10, -1, 1, 6, 18, 21, 32, 32, 71, 94]	[-20, -12, -10, -1, 1, 6, 18, 21, 32, 32, 71, 94]	True
8	8	[-1, 80, 19, 98, 37, 77, 69, -16]	[-16, -1, 19, 37, 69, 77, 80, 98]	[-16, -1, 19, 37, 69, 77, 80, 98]	True
9	14	[48, 84, 60, 54, -1, 90, 30, -14, 44, 53, 29, 76, 11, 62]	[-14, -1, 11, 29, 30, 44, 48, 53, 54, 60, 62, 76, 84, 90]	[-14, -1, 11, 29, 30, 44, 48, 53, 54, 60, 62, 76, 84, 90]	True

10	15	[-10, 1, -46, 64, -27, 99, 57, 93, 93, 70, -45, 49, 75, -27, -9]	[-46, -45, -27, -27, -10, -9, 1, 49, 57, 64, 70, 75, 93, 93, 99]	[-46, -45, -27, -27, -10, -9, 1, 49, 57, 64, 70, 75, 93, 93, 99]	True
11	9	[-45, 12, 71, 55, 51, 22, 56, 17, 87]	[-45, 12, 17, 22, 51, 55, 56, 71, 87]	[-45, 12, 17, 22, 51, 55, 56, 71, 87]	True
12	13	[22, -46, 21, -23, -7, -40, 85, 62, 21, -49, -29, 44, -40]	[-49, -46, -40, -40, -29, -23, -7, 21, 21, 22, 44, 62, 85]	[-49, -46, -40, -40, -29, -23, -7, 21, 21, 22, 44, 62, 85]	True
13	4	[75, -2, 95, 12]	[-2, 12, 75, 95]	[-2, 12, 75, 95]	True
14	15	[97, -41, -21, 69, 68, -40, 76, -7, -36, -50, -17, 79, -5, -34, 54]	[-50, -41, -40, -36, -34, -21, -17, -7, -5, 54, 68, 69, 76, 79, 97]	[-50, -41, -40, -36, -34, -21, -17, -7, -5, 54, 68, 69, 76, 79, 97]	True
15	8	[-46, -29, -44, -33, -21, -7, 79, 95]	[-46, -44, -33, -29, -21, -7, 79, 95]	[-46, -44, -33, -29, -21, -7, 79, 95]	True
16	14	[83, -38, -44, 56, 38, 36, 41, 86, -28, -19, 31, -42, -36, -16]	[-44, -42, -38, -36, -28, -19, -16, 31, 36, 38, 41, 56, 83, 86]	[-44, -42, -38, -36, -28, -19, -16, 31, 36, 38, 41, 56, 83, 86]	True
17	10	[71, 65, 56, 51, 29, 28, 85, 76, 64, 62]	[28, 29, 51, 56, 62, 64, 65, 71, 76, 85]	[28, 29, 51, 56, 62, 64, 65, 71, 76, 85]	True
18	13	[40, 78, 56, 90, 93, 66, -8, 59, 54, 68, 92, 25, -7]	[-8, -7, 25, 40, 54, 56, 59, 66, 68, 78, 90, 92, 93]	[-8, -7, 25, 40, 54, 56, 59, 66, 68, 78, 90, 92, 93]	True
19	14	[7, 77, 80, -48, -29, 47, -2, 71, 17, 37, 63, 6, 31, -49]	[-49, -48, -29, -2, 6, 7, 17, 31, 37, 47, 63, 71, 77, 80]	[-49, -48, -29, -2, 6, 7, 17, 31, 37, 47, 63, 71, 77, 80]	True
20	15	[79, 40, 2, 48, -35, -26, 93, -17, 75, 68, -13, 76, 18, 4, 31]	[-35, -26, -17, -13, 2, 4, 18, 31, 40, 48, 68, 75, 76, 79, 93]	[-35, -26, -17, -13, 2, 4, 18, 31, 40, 48, 68, 75, 76, 79, 93]	True

Table 4.1: Gnome Sort Algorithm Tests

We can see that the algorithm is accurate for both implementations, as all the tests passed. Tests 1 and 2 are edge cases and test for the empty array and an array with only 1 element respectively. In these cases, the algorithm does not change the original array and yields the same result back. The remaining tests all have varying input sizes ranging between 4 and 15 elements to optimize space. These sizes were produced randomly. Additionally, since the randomization for values in the arrays was done with replacement, it is possible for some elements to be repeated such as in tests 4 and 10. It is easy to see that ties are kept in the original position, since the algorithm only swaps elements if the next value is less than the previous. To assess the tests, a function was first implemented to check if the Python result is sorted and then checking the equality of the string representation for both results.

Timings

As mentioned before, the theoretical time complexity of the gnome sort algorithm is $O(N^2)$. Therefore, as the input size increases by 2 times, we expect the runtime to quadruple. Table 5.1 below captures the timing in seconds for both implementations. The data was collected using a similar procedure as the tests in Table 4.1, except in this case, the input sizes are not randomized.

Size	C time	Python time
1	0.003607988	0.000003815
2	0.003489733	0.000003815
4	0.002693892	0.000010967
8	0.004082203	0.000021935
16	0.002831221	0.000033140
32	0.003173113	0.000095844
64	0.002605915	0.000372171
128	0.003154993	0.001439810
256	0.003056288	0.006660938
512	0.004380941	0.027739286
1024	0.004406691	0.084968805
2048	0.009407997	0.357270002
4096	0.038686991	1.304574013
8192	0.128387690	4.858428955
16384	0.398785114	18.592221975
32768	1.720034123	81.569689274
65536	7.761588097	402.078373909

Table 5.1: Gnome Sort Algorithm Timings for both C and Python Implementations

Based on the times for both implementations, the times do appear to be multiplying especially for the larger array sizes. However, it may not be exactly by 4 times due to factors affecting the computer hardware and processes. There may also be some overhead when printing to stdout within C, which is a linear time operation. The table also indicates that Python does have an advantage over C from size 1 up until 128 and the smaller the size, the higher the gap. However, as soon as the size hits 256 and above, C starts to take over.

Figure 5.3 further demonstrates the scale of performance across both implementations.

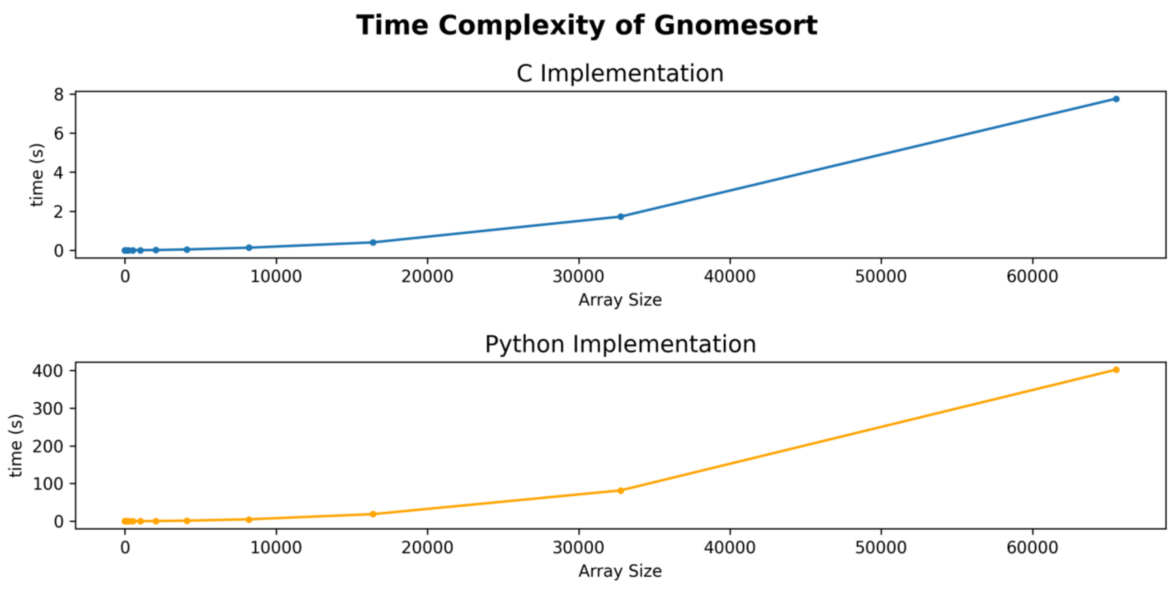

Figure 5.3: Line plots of Gnome sort's time complexity by implementation

As the input size increases, these 2 curves exhibit the same behavior. They both hint at a quadratic or some other polynomial time complexity. However, they vary a lot in terms of scale, which is not surprising, since Python is generally a much slower language compared to C. For example, the highest tested array size takes about 8 seconds to complete using the C implementation of this algorithm. The same operation in Python takes 400 seconds (almost 7 minutes) to complete, so 50 times slower than C. This performance gap will keep growing up to a certain point before flattening out at an asymptote. Since there are varying time scales by the programming languages, a square root transformation was applied on the runtimes to standardize the data and verify the empirical time complexity. We use the square root transformation to test for quadratic time, as this is the inverse transformation.

Figure 5.4 shows a plot of the transformed data for both implementations.

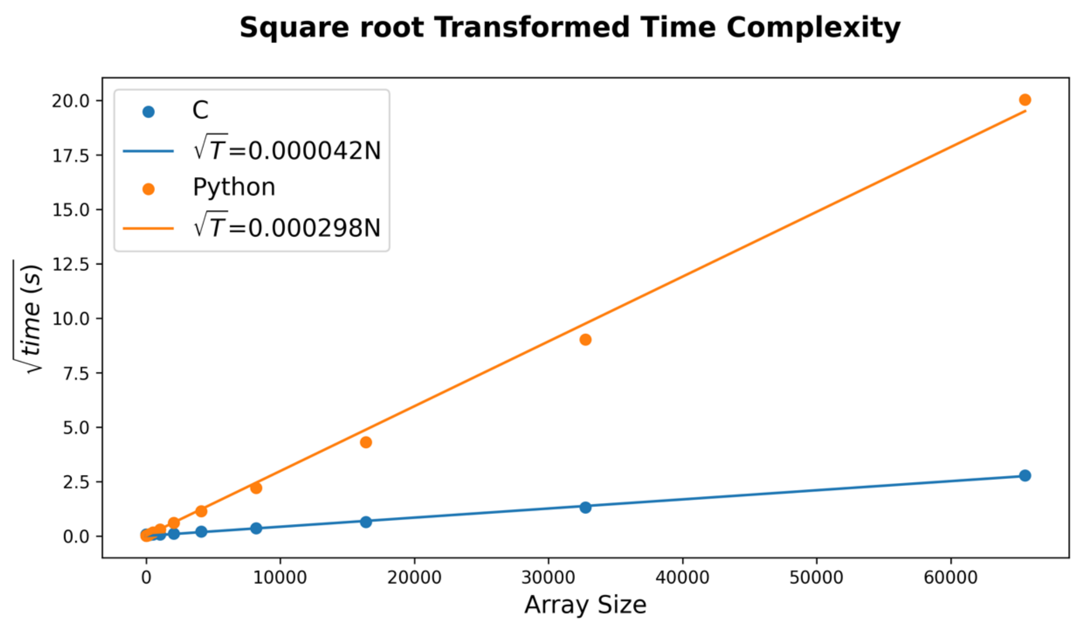

Figure 5.4: Regression plot of the square root transformed runtime data.

The equations of the 2 lines were estimated using a simple linear regression model with no intercept term. We can see that there is an almost perfect linear relationship between the square root runtimes and the array size ($r^2 = 0.995$ for C and $r^2 = 0.997$ for Python). Therefore, this confirms that the empirical time complexity is quadratic, and all estimated run times are being generated correctly. Any variations are simply due to factors involving the system itself. The graph also captures how big the performance gap gets as the data size increases. By inspecting the estimated slopes of the regression models, the square root runtime for Python is approximately 7.1 times higher than C. This means that in the long run as data sizes increase, the algorithm is expected to run about 50 times slower in Python than in C.

Conclusions

In summary, there is not a whole lot of history to gnome sort, and it is quite a simple and effective algorithm when it comes to sorting data. However, just like many other simple sorting algorithms, it is not very efficient. It holds similar runtime as algorithms like insertion and bubble sort, so it should be used for small to medium sized datasets (about 256 elements or higher). After implementing the algorithm in both C and Python, it was determined that while Python has a slightly better performance over C when the data size is very small (roughly 180 elements or lower), C is the better language overall as the data size grows.

References

Azad, Hamid Sarbazi, *Stupid Sort: A new sorting algorithm*, Department of Computing Science Newsletter, University of Glasgow, 599:4, 2 October 2000

"Bogosort or Permutation Sort." *GeeksforGeeks*, GeeksforGeeks, 11 July 2023, www.geeksforgeeks.org/bogosort-permutation-sort/.

Grune, Dick. *Gnome Sort - the Simplest Sort Algorithm*, www.dickgrune.com/Programs/gnomesort.html. Accessed 12 Feb. 2024.

"Gnome Sort." *Altcademy Blog*, Altcademy Blog, 15 June 2023, www.altcademy.com/blog/gnome-sort/.

"Internal Working of the Len() Function in Python." *GeeksforGeeks*, GeeksforGeeks, 2 July 2020, www.geeksforgeeks.org/internal-working-of-the-len-function-in-python/.

Sarbazi, Hamid. *Hamid Sarbazi-Azad's Home Page*, 2000, https://sharif.edu/~azad/.

Xenopax. "Truly Awful Algorithms – Gnome Sort." *Xenopax's Blog*, 26 Apr. 2013, nullwords.wordpress.com/2013/03/28/gnome-sort/.