Contents

[Part 1: Finding the Algorithm, Writing the Code 2](#_Toc9441166)

[Ideation: 2](#_Toc9441167)

[Attachment: 2](#_Toc9441168)

[Analysis: 2](#_Toc9441169)

[Pseudocode: 2](#_Toc9441170)

[Attachment: 2](#_Toc9441171)

[Part 2: Time Complexity 2](#_Toc9441172)

[Simplifying Assumptions: 3](#_Toc9441173)

[Equation for Complexity: 3](#_Toc9441174)

[Part 3: Runtime Graphs 4](#_Toc9441175)

[Update: 5](#_Toc9441176)

* **Part 3:** Program a function, method or class that will track the true runtime of your algorithm. Find the true runtime of your algorithm using arrays of varying sizes (e.g., n = 500, n = 1,500, and n= 2,500) using your new tool. Plot, on a Cartesian plane, the runtime of your algorithm as a function of the size of the input array, n.

# Part 1: Finding the Algorithm, Writing the Code

I first started this by seeing what I would do in my own method prior to doing all the research. I started with ignoring the “in-place” restriction so that I could really understand the pros and cons and have something to compare to.

This is my first work-through:

## Ideation:

1. Count the number of elements in the input, store to n.
2. Create a second array with n index locations, numbered 0 to n.
3. Read through the inputs and place the input serially in reverse order at [i]=n, [i+1]=n-1, [i+2]=n-2 all the way down until [n] = 0
4. Output the array in order, from 0 to n.

## Attachment:

Kirstensreversealgorithm.cpp

## Analysis:

The program that I developed, although correct, is not the most efficient. In fact, it takes double the memory to process, since it has both the input array, and the reversal array stored in two different locations.

After analyzing this, the solution was found to start at either end, and then swap the values. However, both solutions require knowing the size of the input array, (and assigning it to a count variable), so that you can know where the relative “end” is to perform the arrangement.

Pseudocode:*(derived from GeeksforGeeks)*

Input array;

Count sizeof array //to find the end

Set begin = index 0

Set end = index count

    while (begin < end)

// While there is still data between the beginning and end, swap each place

// So if there were 10 elements, 1 would go to 10, and 10 would go to 1, then move onto 2 and 9, 3 and 8, etc.

    {   
// SWAP in place function

        int temp = array[begin];

        array[begin] = array[end];

        array[end] = temp;

        begin++;

        end--;

    }

}

// It doesn’t matter if it’s even or odd, because even will end when begin > end, and if it’s odd then it will end when begin = end

## Attachment:

reversealgorithmiterative.cpp

# Part 2: Time Complexity

To find the time complexity, the code can be broken out, line by line, to see each of the function calls and how long they take. From there, an equation can be derived to plot so that it can be seen when this algorithm becomes inefficient, based on the size of the input, n.

Because the algorithm is an in-place swapping one, we know that the space complexity is independent of n. Only what is needed for the function definition and calls is what is required.

## Simplifying Assumptions:

1. One critical function is to **count** the size of the array.

In order to determine the time complexity for this, one can either assume:

1. The function sizeof actually counts it, or
2. The function sizeof is a pointer to the memory location(s) where the array is stored, and can access the size without performing anything above a read.

Research suggested that for most C++ compilers, that sizeof is a pointer, and therefore has time complexity of O(1).

I did my own research to prove it:

#include <bits/stdc++.h>

using namespace std;

int main()

{

int size = 10000000000000000;

int arr[size];

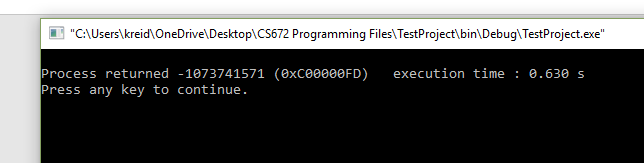
int n = sizeof(arr) / sizeof(arr[0]);

cout<< n;

return 0;

}

I reported out the runtimes from my IDE:



And then plotted them, changing N to go from 100 to 10000000000000000 (I tried each twice to be sure).

|  |  |  |
| --- | --- | --- |
| n | t | t2 |
| 100 | 0.631 | 0.039 |
| 10000 | 0.038 | 0.048 |
| 10000000 | 0.623 | 0.599 |
| 1000000000 | 0.627 | 0.623 |
| 10000000000000 | 0.625 | 0.6 |
| 1E+16 | 0.598 | 0.63 |

From this test, I can see that the time complexity is **approximately stable**. Therefore, I will simplify the count function to O(1).

## Equation for Complexity:

So let’s review the function again, and go line by line:

#include <bits/stdc++.h> [O(1)]

using namespace std; [O(1)]

void reverseArray(int arr[], int start, int end)

{ while (start < end)

{ int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;}}

Here it can be seen that if n = 100, the while loop will repeat 50 times. Linearly it will progress that for every 2n, the loop will occur once. Each of the assignments in the swap (arr[start] = arr[end]) is a simple declaration that happens every loop, so only a constant complexity for the internal functions.

Therefore, the total tome complexity for this function is linear, so O(n/2). With infinite summation, this can be simplified to ½ O(n), which can be further reduced to O(n).

void printArray(int arr[], int size)

{for (int i = 0; i < size; i++)

cout << arr[i] << " ";

cout << endl;}

Similarly, the function structure here is a loop based on input size n that repeats simple output statements; therefore, the time complexity is O(n), because it will loop n times.

int main()

{ int arr[] = {11, 75, 3231, 4, 512, 61}; [O(1)]

int n = sizeof(arr) / sizeof(arr[0]); [O(1)]

printArray(arr, n); [O(n)]

reverseArray(arr, 0, n-1); [O(n/2)]

printArray(arr, n); [O(n)]

return 0;} [O(1)]

When we summarize all of these together, we get:

Time Complexity (g) = c1[O(1)] + c2[O(1)] + c3[O(n)] + c4/2[O(n)] + c5[O(n)] + c6[O(1)] =

(g) = {c1+c2+c6} [O(1)] + {c3 + c4/2 + c5} [O(n)] = A [O(1)] + B[O(n)] where A = (C1 + C2 + C6) and B = (C3 + C4/2 + C5) and each are constants.

The space complexity is still O(1), for every n.

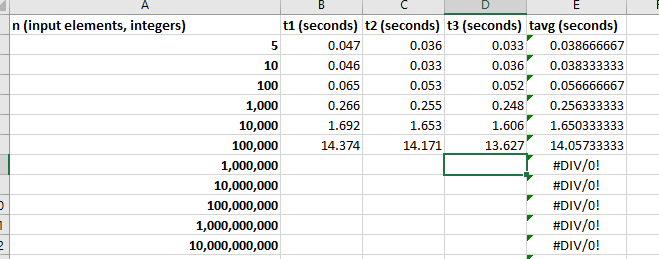
# Part 3: Runtime Graphs

In order to calculate the actual runtimes and determine the coefficients C1 and C2, the program and its runtimes were plotted for 20 runs. Then, systematic equation analysis could be utilized to find the regression curve and plot it. **Note: My codeblocks IDE already came with a runtime, so I added the runtime code later, but didn’t plot it again. It was still linear 😊**

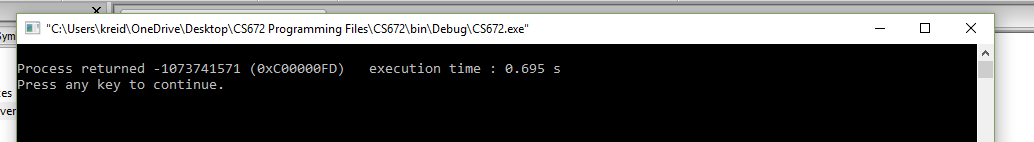
## Update:

At first, I planned on plotting n = 5, 10, 100, 100.. all the way to n = 1\*1030. What I didn’t factor in was that eventually the buffer would overload and I couldn’t perceive or gather time passage. 😊

As I was plotting this:



I got the following when n = 1,000,000.



SO: In order to still get good regression profiling, I added a few more *middle* points. The summary is below:

Therefore, to find the regression coefficients, an excel program was used:

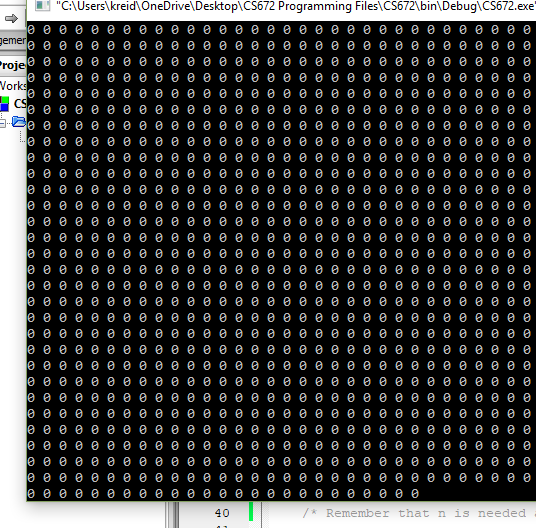
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n (input elements, integers)** | **t1 (seconds)** | **t2 (seconds)** | **t3 (seconds)** | **tavg (seconds)** |
| **5** | 0.047 | 0.036 | 0.033 | 0.038666667 |
| **10** | 0.046 | 0.033 | 0.036 | 0.038333333 |
| **100** | 0.065 | 0.053 | 0.052 | 0.056666667 |
| **1,000** | 0.266 | 0.255 | 0.248 | 0.256333333 |
| **10,000** | 1.692 | 1.653 | 1.606 | 1.650333333 |
| **100,000** | 14.374 | 14.171 | 13.627 | 14.05733333 |
| **20** | 0.051 | 0.035 | 0.037 | 0.041 |
| **500** | 0.143 | 0.143 | 0.14 | 0.142 |
| **6,666** | 1.164 | 1.145 | 1.152 | 1.153666667 |
| **19,875** | 3.047 | 2.988 | 3.02 | 3.018333333 |
| **200,001** | 28.064 | 27.1 | 27.697 | 27.62033333 |
| **890** | 0.234 | 0.449 | 0.219 | 0.300666667 |
| **6,994** | 1.249 | 1.225 | 1.251 | 1.241666667 |
| **89,335** | 12.698 | 12.589 | 12.964 | 12.75033333 |
| **300,001** | 39.742 | 40.048 | 39.453 | 39.74766667 |

And here are the plots:

Where x = n, and y = g (or t).

From our original equation, A [O(1)] + B[O(n)], we confirm that the worst case time complexity is linear, with coefficients (A = .2367, B = 0.0001) and R2 (loosely coupled as accuracy) value of 99.9%.

One thing to note: while compiling, the assignment of int arr[n]; allocates a supposedly random set of integers. For small numbers, this works well. However, at large instances of n, I noticed that the majority of them were 0:



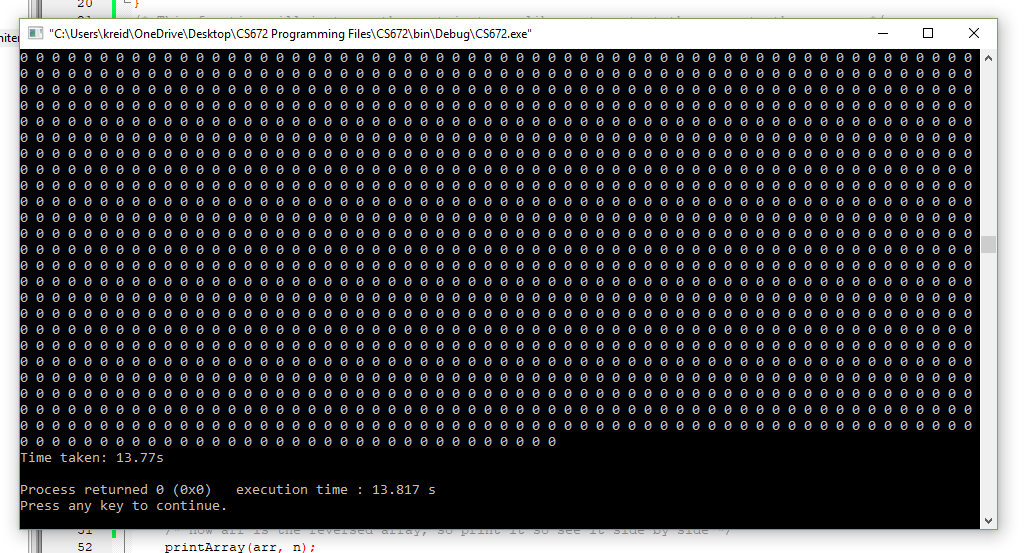


I will assume that this assignment didn’t affect the time complexity, as an integer of 0 takes up the same amount of space as 3242, but it is interesting to note.

However, 350,000 input integers is hardly an “infinite series” comparison. Therefore, to truly understand the relationship between the input and the time complexity, a program was developed to change the input and capture the output as a function.

Note that declaring n input variable and the function to capture time will affect the regression slightly, but only in coefficient factor and offset.

I tested an initial program, and found that above a certain threshold, ~2000, that the timing was approximately correct.



Here is the code (from Stack Overflow, n.d.):   
You could also manually modify your C program to instrument it using the clock() library function, like so:

#include <time.h>

int main(void) {

clock\_t tStart = clock();

/\* Do your stuff here \*/

printf("Time taken: %.2fs\n", (double)(clock() - tStart)/CLOCKS\_PER\_SEC);

return 0;

}

Also, you can use the chrono library, but my IDE didn’t support it, so I figured simpler was better.

## Comparison:

I re-ran my old code to see it’s time complexity and got the following:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n (input elements, integers)** | **t1 (seconds)** | **t2 (seconds)** | **t3 (seconds)** | **tavg (seconds)** |
| **5** | 0.045 | 0.034 | 0.038 | 0.039 |
| **10** | 0.054 | 0.043 | 0.041 | 0.046 |
| **100** | 0.067 | 0.065 | 0.062 | 0.064666667 |
| **1,000** | 0.381 | 0.373 | 0.361 | 0.371666667 |
| **10,000** | 3.105 | 3.028 | 3.033 | 3.055333333 |
| **100,000** | 10.016 | 9.856 | 10.346 | 10.07266667 |
| **20** | 0.047 | 0.038 | 0.037 | 0.040666667 |
| **500** | 0.217 | 0.2 | 0.201 | 0.206 |
| **6,666** | 2.232 | 2.279 | 2.298 | 2.269666667 |
| **19,875** | 3.88 | 3.81 | 3.847 | 3.845666667 |
| **200,001** | 16.718 | 17.804 | 17.308 | 17.27666667 |
| **890** | 0.335 | 0.335 | 0.33 | 0.333333333 |
| **6,994** | 2.285 | 2.289 | 2.296 | 2.29 |
| **89,335** | 9.365 | 9.269 | 9.285 | 9.306333333 |
| **300,001** | 25.133 | 24.919 | 25.858 | 25.30333333 |

And when compared:

(Blue is the actual swap algorithm, orange is mine that uses 2 arrays).

At large values, the in-place swapping algorithm is less efficient than the double-array method. However, the size complexity of the in-place is constant, while with the double array, it is linear with respect to n.

**References:**

1. Stack Overflow (n.d.) *How do sizeof(arr) and sizeof(arr[0]) work?* Retrieved from <https://stackoverflow.com/questions/33523585/how-do-sizeofarr-sizeofarr0-work>
2. Cplusplus (n.d.) *Functions*. Retrieved from <http://www.cplusplus.com/doc/tutorial/functions/>
3. Tutorials Point (n.d.) *Passing arrays to functions in C++.* Retrieved from <https://www.tutorialspoint.com/cplusplus/cpp_passing_arrays_to_functions.htm>
4. Cplusplus (n.d.) *Arrays*. Retrieved from <http://www.cplusplus.com/doc/oldtutorial/arrays/>
5. Geeks for Geeks (n.d.) *Write a program to reverse an array or string*. Retrieved from <https://www.geeksforgeeks.org/write-a-program-to-reverse-an-array-or-string/>
6. Mejia, A. (2018, Apr 5). *8 time complexities that every programmer should know.* Retrieved from <https://adrianmejia.com/blog/2018/04/05/most-popular-algorithms-time-complexity-every-programmer-should-know-free-online-tutorial-course/>
7. Tanwar, S. (2018, Mar 26) *Let’s simplify algorithm complexities!* freeCodeCamp. Retrieved from <https://medium.freecodecamp.org/lets-simplify-algorithm-complexities-25e75f37d03f>
8. Khatri, S. (2018, Aug 21). *Analyzing complexity of code through Python*. DataCamp. Retrieved from <https://www.datacamp.com/community/tutorials/analyzing-complexity-code-python>
9. Stack Overflow (n.d.) *Calculating Execution time in C++.* Retrieved from <https://stackoverflow.com/questions/876901/calculating-execution-time-in-c>