Andrew Sivaprakasam | Warm-Up 1 Write-up
GitHub Repo: `https://github.com/sivaprakasaman/Numerical_Methods_BME/`

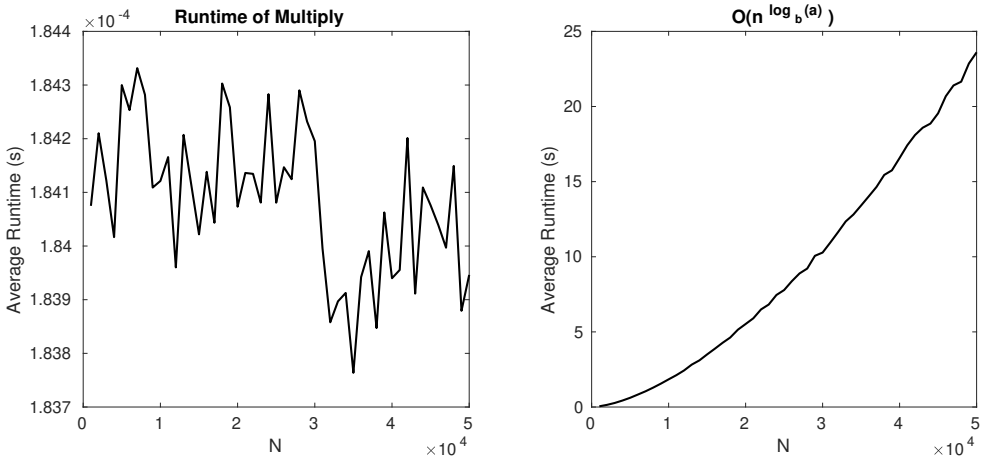
1 Divide-and-Conquer for Integer Multiplication

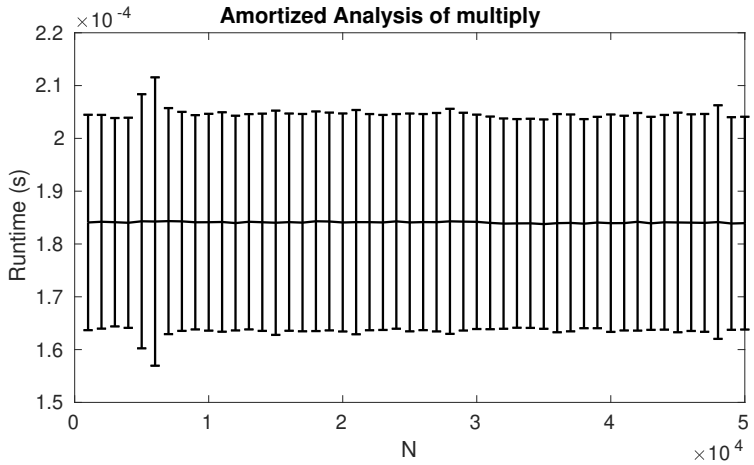
After implementing the provided `multiply` algorithm, correctness was verified using the provided multiplications.

Furthermore, runtime analysis was conducted, and no visible trend was observed, even after 10,000 repetitions of the given square product computations from 1,000 to 50,000. This may have been due to variations caused by the functions used to convert binary to decimal `bin2dec` and `dec2bin` or an implementation issue.

The Master Theorem states that in a problem where a subproblems of size n/b that combines these subproblems with $O(n^d)$ time-complexity, the final time complexity will be $O(n^{log_b(a)})$ if $d < log_b(a)$. This is the case in the `multiply` function since $log_2(3) \approx 1.6$.

I plotted this trend as a function of N alongside the actual runtimes. Unfortunately, there does not appear to be a correlation in the two.





2 Sorting Algorithms

2.1 Typset Sorting Algorithm Statements

Note: My typset algorithms are *zero-based*– they assume indexing starts at zero, not the case in Matlab.

Bubble Sort: Typset Algorithm. Worst/Average Case = $O(n^2)$, Best Case = $O(n)$

---

function bubbleSort($a[1, ..., k]$)

input : An unsorted array, a
output: The sorted array

Iterate through each element i in a, swapping it with the element prior if element i-1 is greater than element i

swapped = **true**;

while *swapped* **do**
    swapped = **false**;
    for $i \leftarrow$ *1* **to** *k-1* **do**
        if $a[i-1] > a[i]$ **then**
            swap: $a[i-1] \leftrightarrow a[i]$;
            swapped = **true**;
        end
    end
end

Insertion Sort: Typset Algorithm. Worst/Average Case = $O(n^2)$, Best Case = $O(n)$

---

function insertionSort($a[1, ..., k]$)
input : An unsorted array, a
output: The sorted array

Incrementally traverse through each element i in a, sliding a[i] left through the sorted array, until it is ranked appropriately

$i = 1$;

while $i < k$ **do**
    $j = i$;
    while $j > 0$ *and* $a[j-1] > a[j]$ **do**
        swap: $a[j] \leftrightarrow a[j-1]$;
        $j \leftarrow$ j - 1;
    end
    $i \leftarrow i + 1$;
end

Selection Sort: Typset Algorithm. Worst/Average/Best Case = $O(n^2)$

function selectionSort($a[1, ..., k]$)
input : An unsorted array, a
output: The sorted array

for $i \leftarrow$ *0* **to** *k-1* **do**
    starting with element i, compare i against elements ahead. If there is a new minimum, swap it with element i
    tempMin = i;
    for $j \leftarrow i + 1$ **to** k **do**
        if $a[j] < a[tempMin]$ **then**
            tempMin = j;
        end
    end
    if *tempMin does not equal i* **then**
        swap: $a[i] \leftrightarrow a$[tempMin];
    end
end

2.2 Time and Space Complexity Justification

Name	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)	Stable
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)	Unstable
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)	Stable
Merge Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)	Stable
Quick Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))	Unstable
Heap Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)	Unstable
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)	Stable
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)	Stable

Time Complexity:

Bubble Sort and Insertion Sort are the same in terms of their best, average, and worst-case time complexities. Both algorithms allow a single pass of iterations through every element in the array in the best case (already sorted array), $\Omega(n)$. In the average and worst cases for these algorithms, there will need to be some-sort of re-iteration through the array to accommodate out-of-order elements.

Selection Sort technically has worse best-case time-complexity than both Bubble Sort and Insertion Sort, because it requires a nested iteration, even in an already sorted array. Merge Sort has the best average and worst-case time complexities, since its recursive nature

allows for solving of smaller sub-problems, to get to $O(nlog(n))$. However, its best case time-complexity is still worse than Bubble/Insertion sort because it does not change its necessary iterations even in a sorted array.

Space Complexity:

Bubble, Insertion, and Selection Sort all have space complexity of $O(1)$, because the only additional space needed their operations is the storage of a single element to be swapped. Merge sort, however, has space complexity of $O(n)$ because it requires N elements of storage as it breaks up the array into smaller arrays to sort and merge together.

***Note:** Time-complexity does not always translate to efficiency, since the number of elements swapped is not accounted for. Though Bubble Sort, Insertion Sort, and Selection Sort have the same or similar time-complexities, the number of swaps will be higher in Bubble Sort, decreasing its efficiency.

2.3 Experimental Testing of Amortized Analysis

The aforementioned sorting algorithms were implemented in Matlab, and the runtimes were measured. Bubble Sort, though slightly lower in best-case time-complexity than Selection and Insertion Sort, performed worse than those two, presumably due to the inefficiency in the necessary number of swaps occuring during sorting.

Additionally, Merge Sort did not perform as expected. It should have been the fastest of the implemented algorithms. *This may be due to an implementation issue or a Matlab issue.*

Two additional sets of 10 arrays (dim = 500 and 100) were generated to further assess the runtimes.