FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION  
ITMO UNIVERSITY

Report

on practical task No. 1

“Experimental time complexity analysis”

Performed by

Etietop Abraham

Academic Group J4134c

Accepted by

Dr. Petr Chunaev

St. Petersburg

2021

**Goal**

Observe and study the running times of executing different algorithms on randomly varying datasets of non-negative integers.

**Problem Formulation**

***I. Generate an n-dimensional random vector* 𝒗 = [𝑣%, 𝑣', ... , 𝑣)] *with non-negative elements. For* 𝒗*, implement the following calculations and algorithms:***

*Random vector of size N. V = [v1,v2, … vn]*

*N = 2000*

*Runtime = 0.0*

*Time Complexity = Worst-case in Big O Notation*

*1. F(v) = A Constant Function*

*// Constant Function that adds two integers*

1. *Input = integer a, b*
2. *Start = start time*
3. *Output = a + b*
4. *Stop = end time*
5. *Runtime = Stop - Start*
6. *Return Output, Runtime*

*2. F(v) = Sum of elements in V*

*// This function takes a list and returns the sum of all elements*

1. *Input = List of random vectors*
2. *Sum = 0*
3. *Start = start time*
4. *For counter in List Size*
   * 1. *Sum = sum + List[counter]*
5. *Stop = stop time*
6. *Runtime = Stop - Start*
7. *Return Sum, Runtime*

*3. F(v) = Product of elements in V*

*// Similar with 2.4.i, difference is multiplication*

*Product = 1*

*Start = start time*

*For counter in List Size*

*Product = product \* List[counter]*

*Stop = stop time*

*Runtime = Stop - Start*

*Return Product, Runtime*

*4. Polynomial* ∑nk=1 Vk X^(k-1)

*// Evaluate with Loops*

*Input = List of coefficients, and value of x in term*

*Result = 0*

*Start = start time*

*For counter in List Size*

*Result = Result + List[counter] \* x^(len(list)-counter)*

*Stop = stop time*

*Runtime = Stop – Start*

*Return Result, Runtime*

*// Evaluate with Horner’s Method*

1. *Input = List of coefficients, and value of x in term*
2. *Initialize the value of the result to the coefficient of the first term or index*
   1. *Result = List[0]*
3. *Start = start time*
4. *For counter in range(1, len(list))*
   1. *Result = Result \* x + List[counter]*
5. *Stop = stop time*
6. *Runtime = Stop – Start*
7. *Return Result, Runtime*

*5. Bubble Sort of the elements of V*

*//By traversing through all list in V*

*//And checking if next list is lower*

*//If so switch*

1. *Input = List of non-negative integers*
2. *Start = start time*
3. *For i in range(len(List))*
   1. *For j in range(0, n-i-1)*
      1. *If list[j] > list[j+1]:*
         1. *List[j], list[j+1] = list[j+1], list[j]*
4. *Stop = stop time*
5. *Runtime = Stop – Start*
6. *Return Runtime*

*6. Quick Sort of the elements of V*

*//Quick sort is a divide and conquer method*

*//It selects an element as pivot*

*//And partitions list around pivot.*

*Input = List of non-negative integers*

*Start = start time*

*Pivot = List[0], First element*

*Sort Pivot*

*Elements smaller than x before x, and greater elements after x*

*Runtime = Stop - Start*

*Stop = stop time*

*Return Runtime*

*7. Timsort of the elements of V*

*//Timsort is based on Insertion and Merge Sort*

*//Used in Python’s sorted() & sort() function*

*//Sort small pieces with Insertion Sort*

*//Then merge the pieces using a merge of merge sort*

***II. Generate random matrices* 𝐴 *and* 𝐵 *of size* 𝑛 × 𝑛 *with non-negative elements. Find the usual matrix product for* 𝐴 *and* 𝐵**

*//Multiply an nXn matrix A, and B*

*Input = Random Matrix A, and B of size n X n*

*Start = start time*

*For I in range(len(A), len(B)*

*For j in range(len(B)[0])*

*For k in range(len(B))*

*Result[i][j] += A[i][k] \* B[k][j]*

*Stop = stop time*

*Runtime = Stop - Start*

*Return Result, Runtime*

***II. Generate random matrices* 𝐴 *and* 𝐵 *of size* 𝑛 × 𝑛 *with non-negative elements. Find the usual matrix product for* 𝐴 *and* 𝐵**

***III. Describe the data structures and design techniques used within the algorithms****.*

**Brief Theory**

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., a step-by-

step-specific instruction that accepts a legitimate input and produces the required output in a finite amount of time.

Input >> ALGORITHM >> Output

First, before designing an algorithm, one must know whether the problem can be solved and

how difficult the problem is, before journeying on finding a suitable algorithm and making it

efficient.

The computational complexity/performance of an algorithm depends on how much data is

thrown at it, and we analyze an algorithm in order to understand how an algorithm scales, as

we throw more data at it.

It is also very important to define the instances of performance for an algorithm, the upper limit or worst-case scenario, and the lower limit or best-case scenario.

The time complexity of an algorithm measures the running time function T(n), which returns

the time is taken for an algorithm to complete its operation.

E.g., T(n) = A\*N^2 + B\*N + C where A, B & C are unspecified constants.

The worst-case running time of an algorithm is represented using the Big O Notation, while

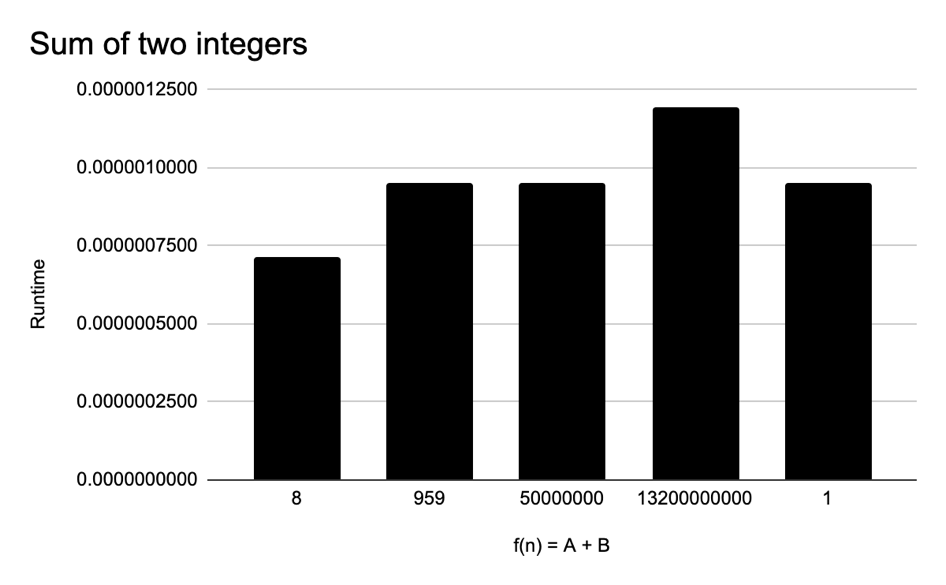
the best case is represented by Omega. Running time T(n) of the above formula, the worst-case running time overall is denoted as O(n^2), and the best-case scenario is denoted by Ω(n^2).

The tendency of an algorithm to scale is determined by the highest order term within the formula/method/algorithm.

**Results**

1. *F(v) = A Constant Function*

|  |  |  |  |
| --- | --- | --- | --- |
| F(n) = Add(a, b) | | |  |
| a | b | output | runtime |
| 4 | 4 | 8 | 0.0000007153 |
| 414 | 545 | 959 | 0.0000009537 |
| 17000000 | 33000000 | 50000000 | 0.0000009537 |
| 7700000000 | 5500000000 | 13200000000 | 0.0000011921 |
| 1 | 1 | 1 | 0.0000009537 |

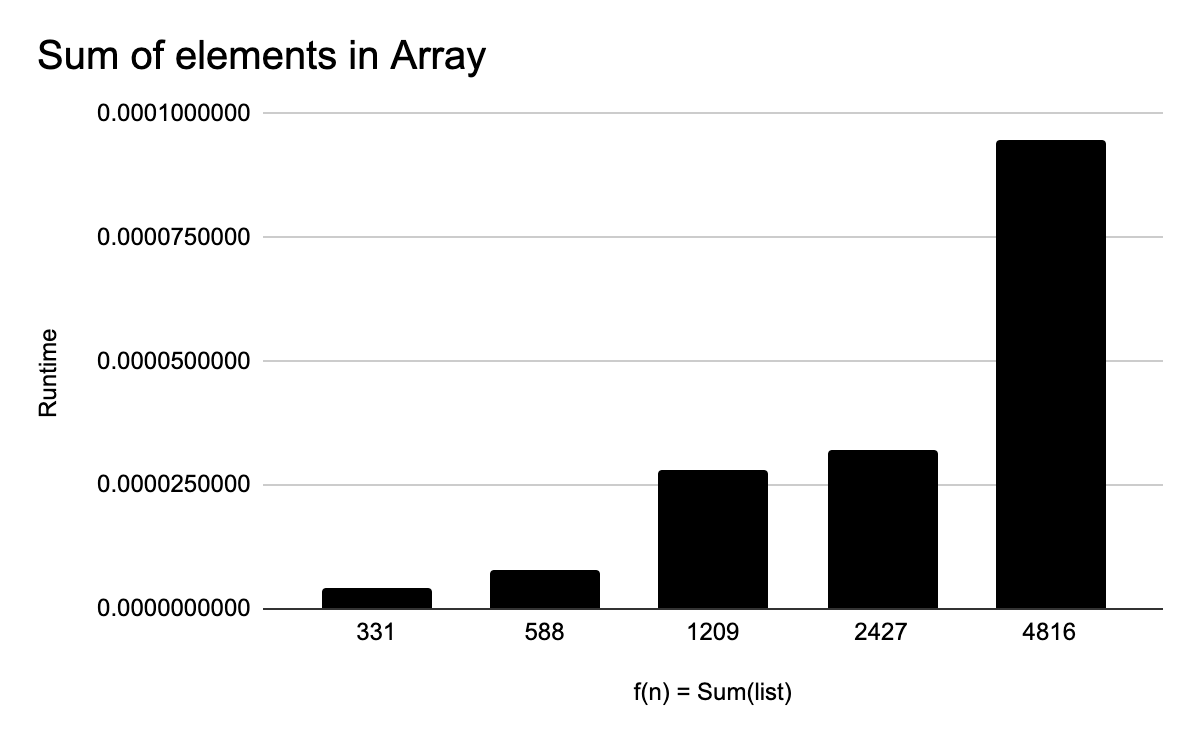


*Complexity Analysis.*

*The best- & worst-case running time is Θ (1), regardless), regardless of the size of integers, execution completes of the size of integers, execution completes in order of a constant.*

1. *F(v) = Sum of elements in V*

|  |  |  |
| --- | --- | --- |
| 𝑓(𝒗) = ∑ 𝑣 (the sum of elements); | | |
| n | output | runtime |
| 100 | 331 | 0.0000040531 |
| 200 | 588 | 0.0000078678 |
| 400 | 1209 | 0.0000278950 |
| 800 | 2427 | 0.0000319481 |
| 1600 | 4816 | 0.0000948906 |

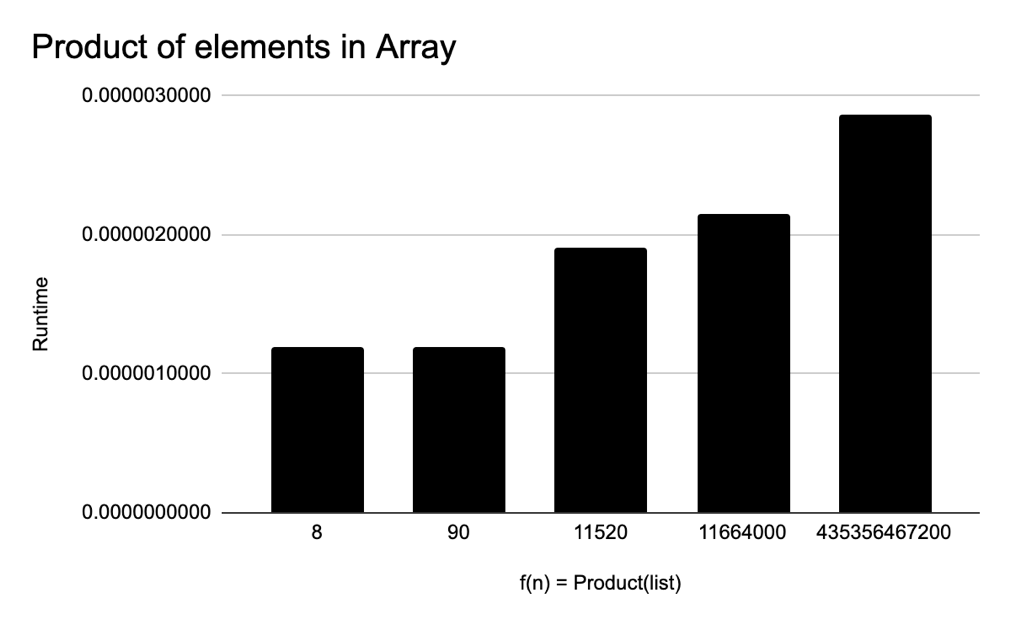


*Complexity Analysis.*

*The worst-case running time is O (n), as the size of the list increases, the time taken to complete execution. From left to right, is the performance for list sizes ranging from 100 to 1600, with completion time more than doubling for the larger datasets.*

1. *F(v) = Product of elements in V*

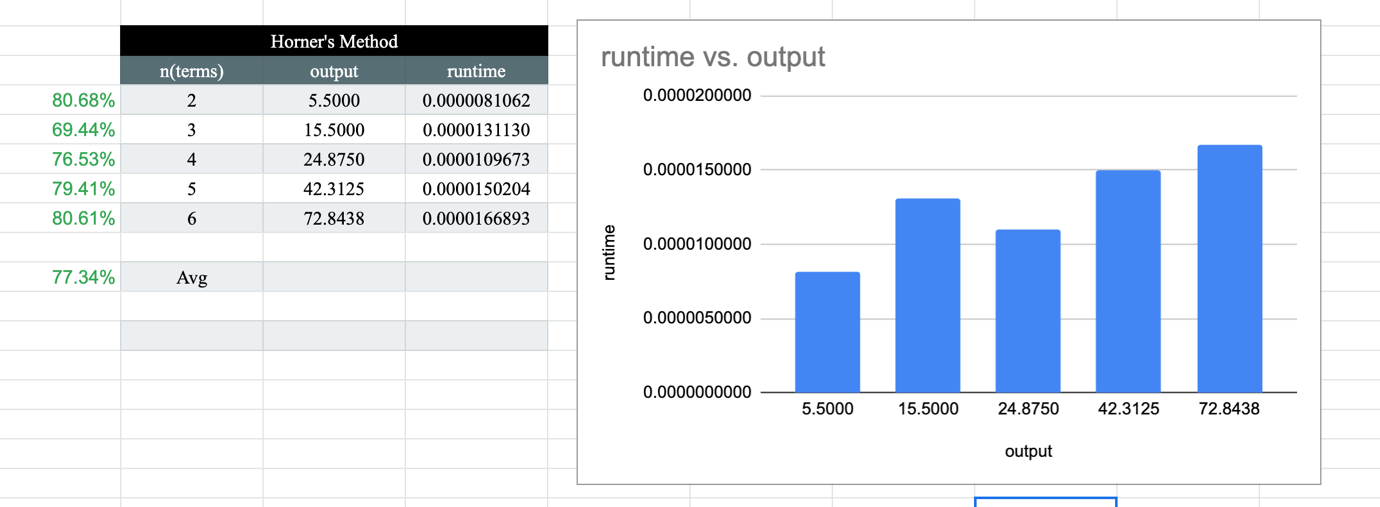
|  |  |  |
| --- | --- | --- |
| 𝑓(𝒗) = \* 𝑣 (the product of elements); | | |
| n | output | runtime |
| 2 | 8 | 0.0000011921 |
| 4 | 90 | 0.0000011921 |
| 8 | 11520 | 0.0000019073 |
| 16 | 11664000 | 0.0000021458 |
| 32 | 435356467200 | 0.0000028610 |



*Complexity Analysis.*

*The worst-case running time is O (n), as the size of the list increases, the time taken to complete execution. From left to right, is the performance for list sizes ranging from 2 to 32, with completion time more than doubling for the larger datasets.*

1. *F(v) = Polynomial* ∑nk=1 Vk X^(k-1)

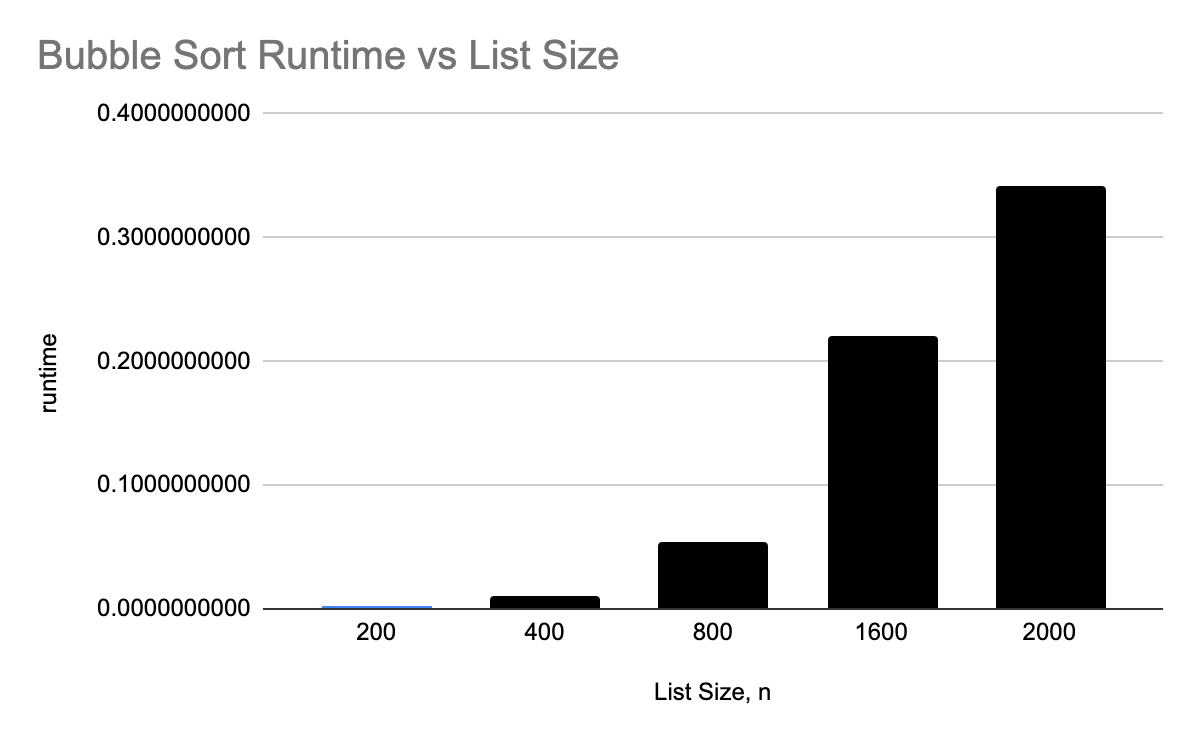
**

*Complexity Analysis.*

*First off, results from Horner’s method on average* ***executed 77% times faster*** *than the loop evaluating the polynomial. Evaluating with loops scales on the order of O(n2), and Horner’s method on the order of O(n)*

1. *F(v) = Bubble Sort of the elements of V*

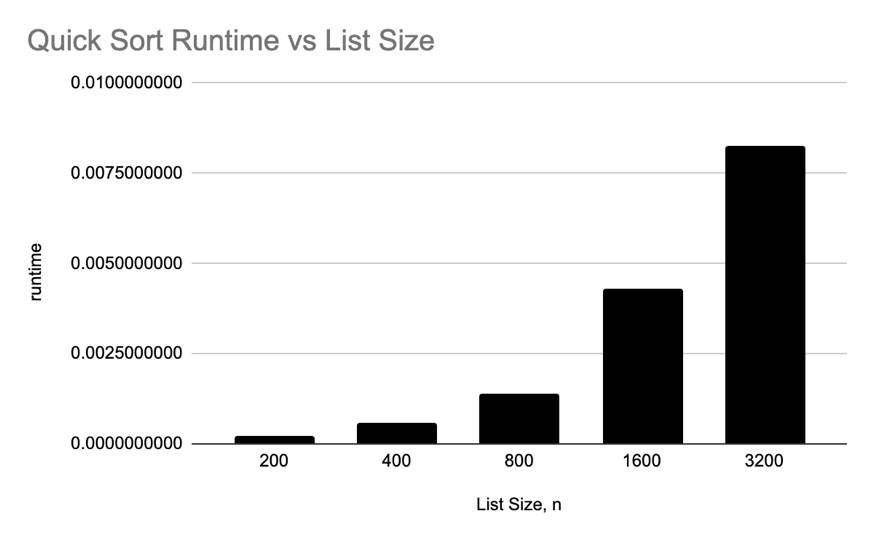
|  |  |
| --- | --- |
| Bubble Sort | |
| n | runtime |
| 200 | 0.0026278496 |
| 400 | 0.0110151768 |
| 800 | 0.0535857677 |
| 1600 | 0.2198369503 |
| 2000 | 0.3409359455 |



*Complexity Analysis.*

*The Worst-case occurs when elements are sorted in descending order, in the order of O(n2), in which case, iterating through all the elements in the list is required to sort list.*

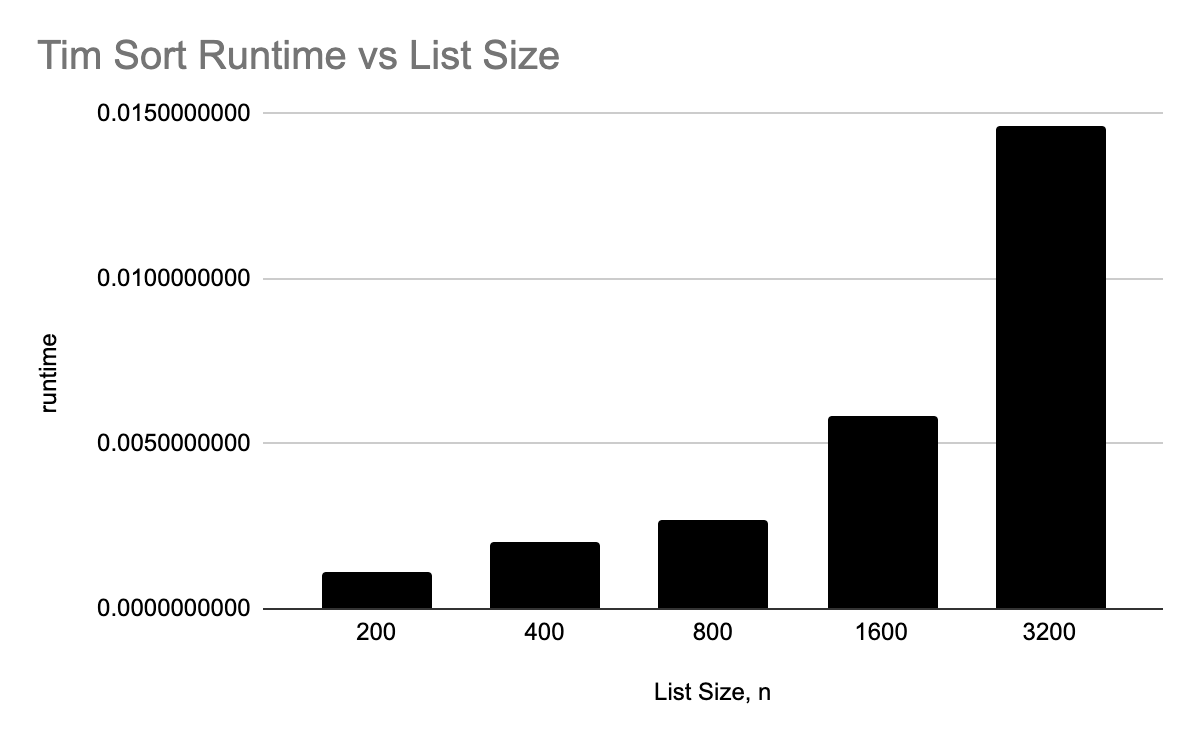
|  |  |
| --- | --- |
| Quick Sort | |
| n | runtime |
| 200 | 0.0002348423 |
| 400 | 0.0005910397 |
| 800 | 0.0014040470 |
| 1600 | 0.0043120384 |
| 3200 | 0.0082538128 |

1. *F(v) = Quick Sort of the elements**Complexity Analysis.*

*In general, time taken by Quicksort is T(n) = T(k) + T(n-k-1) +* Φ*(n), and worst-case is T(n) = T(n-1) +* Φ*(n)*

1. *F(v) = Tim Sort of the elements*

|  |  |
| --- | --- |
| Tim Sort | |
| n | runtime |
| 200 | 0.0011126995 |
| 400 | 0.0020308495 |
| 800 | 0.0026643276 |
| 1600 | 0.0058569908 |
| 3200 | 0.0146250725 |

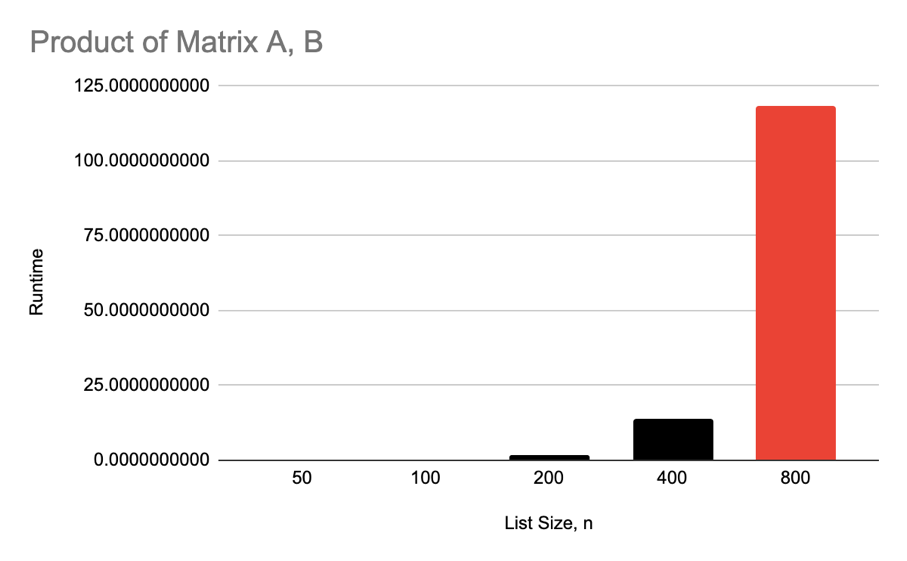


*Complexity Analysis.*

*Best case O(n), and worst-case O(n\*log(n)). Taking 77% less time to process a list of 2000 elements when compared with Quick sort.*

***II*** *F(v) = Product of Matrix A and B*

|  |  |  |
| --- | --- | --- |
| Product of Matrix A, B (nXn) | | |
| n | m | runtime |
| 50 | 50 | 0.0274517536 |
| 100 | 100 | 0.2072679996 |
| 200 | 200 | 1.6804139614 |
| 400 | 400 | 13.8756799698 |
| 800 | 800 | 118.4307587147 |



*Complexity Analysis.*

*Looping three times in other to count through rows in matrix A, and B leads to higher time to execute. Scaling in the order of O(n3).*

***II*** *Data structure and design techniques used with the algorithms*

*The* ***Sum of elements and product of elements function*** *uses a linear list data structure to hold a list of integers. And technique used to compute the result is a simple loop iteration.*

*Polynomial function evaluation with Horner’s method uses a linear data structure, like in sum/product, however, the design technique in the algorithm need not loop through all data, but, initialize the result with the coefficient of the first element, and repeatedly multiply the result with x, and add next coefficient to result.*

***Bubble****, Quick, and Timsort use linear list data structures, but different techniques. Bubble, by traversing through all lists in checking if the next list is lower or higher, If so switch.* ***Quick sort****, on the other hand, uses a divide and conquer method by selecting an element as a pivot, and partitioning a list around the pivot. Finally, Timsort is based on Insertion and Merge Sort, sorting small pieces with Insertion Sort, then merges the pieces using a merge of merge sort*

**Conclusions**

*Executing a task requires finding a balance between time resource and space requirements. Different techniques of solving a problem tackle to improve on time with respect to growth/scaling of the function, as the size of input increases.*

**Appendix**

[*https://github.com/eti-etop/algorithm-time-complexity*](https://github.com/eti-etop/algorithm-time-complexity)

*Result on Google Sheets*

[*Geeks for Geeks*](https://www.geeksforgeeks.org/)