

**Name Surname:** Gülendam Buket Gündüz

**Student Number:** 21328026

**Aim of the Experiment**

In this experiment, it is expected from us to solve some problems like finding maximum element of an array or multiplying by nxn matrix etc. with optimized algorithms and analyse them. We use Java Programming Language for implementing the algorithms.

**Analysis of Algorithms**

In computer science, people use algortihms to solve difficult problems or process lots of data. But there are two question marks when one tries to solve a problem:

* How long will my program take?
* Why does my program run out of memory?

The answers depend on many factors such as properties of the particular computer being used, the particular data being processed or the particular program that is doing the job. All these factors give us lots of imformation to analyze.

The analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to [execute](https://en.wikipedia.org/wiki/Computation) them. Most [algorithms](https://en.wikipedia.org/wiki/Algorithm) are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations ([space](https://en.wikipedia.org/wiki/Space_complexity) complexity). Analysis of algorithms helps us to understand theoretical basis and to compare algorithms that is used to solve same problems. If an algorithm use lots of space and its running time is too long, it is not appropriate approach. The best algorithm is the algorithm which uses minimum space and time.

To study the cost of running time, we study our programs themselves via the scientific method, the commonly accepted body of techniques universally used by scientists to develop knowledge about the natural world. We also apply *mathematical analysis* to derive concise models of the cost.

**Scientific method:**

 Our approach is the scientific method, and it involves the following 5 step approach.

* Observe some feature of the natural world.
* Hypothesize a model that is consistent with the observations.
* Predict events using the hypothesis.
* Verify the predictions by making further observations.
* Validate by repeating until the hypothesis and observations agree.

**Mathematical Analysis:**

Total running time is basicly equal to multiplactionof sum of cost and frequency for all operations.

* Need to analyze program to determine set of operations.
* Cost depends on machine, compiler.
* Frequency depends on algorithm, input data.

In this experiment we focus on running times of algorithms, so it is expected from us to follow several steps to calculate time complexity of algorithms:

* Implement the algorithm completely.
* Determine the time required for each basic operation.
* Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
* Develop a realistic model for the input to the program.
* Analyze the unknown quantities, assuming the modelled input.
* Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

I have used the following example, based on information provided to us.

Unit Cost Times

i=1; c1 1

sum = 0; c2 1

while (i <= n) { c3 N+1

j=1; c4 N

while (j <= n) { c5 N\*(N+1)

sum = sum + i; c6  N\*N

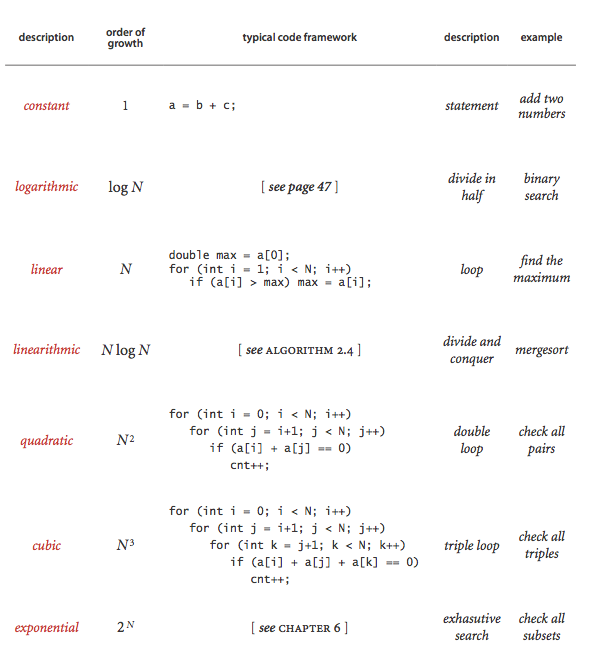
j = j + 1; c7 N\*N

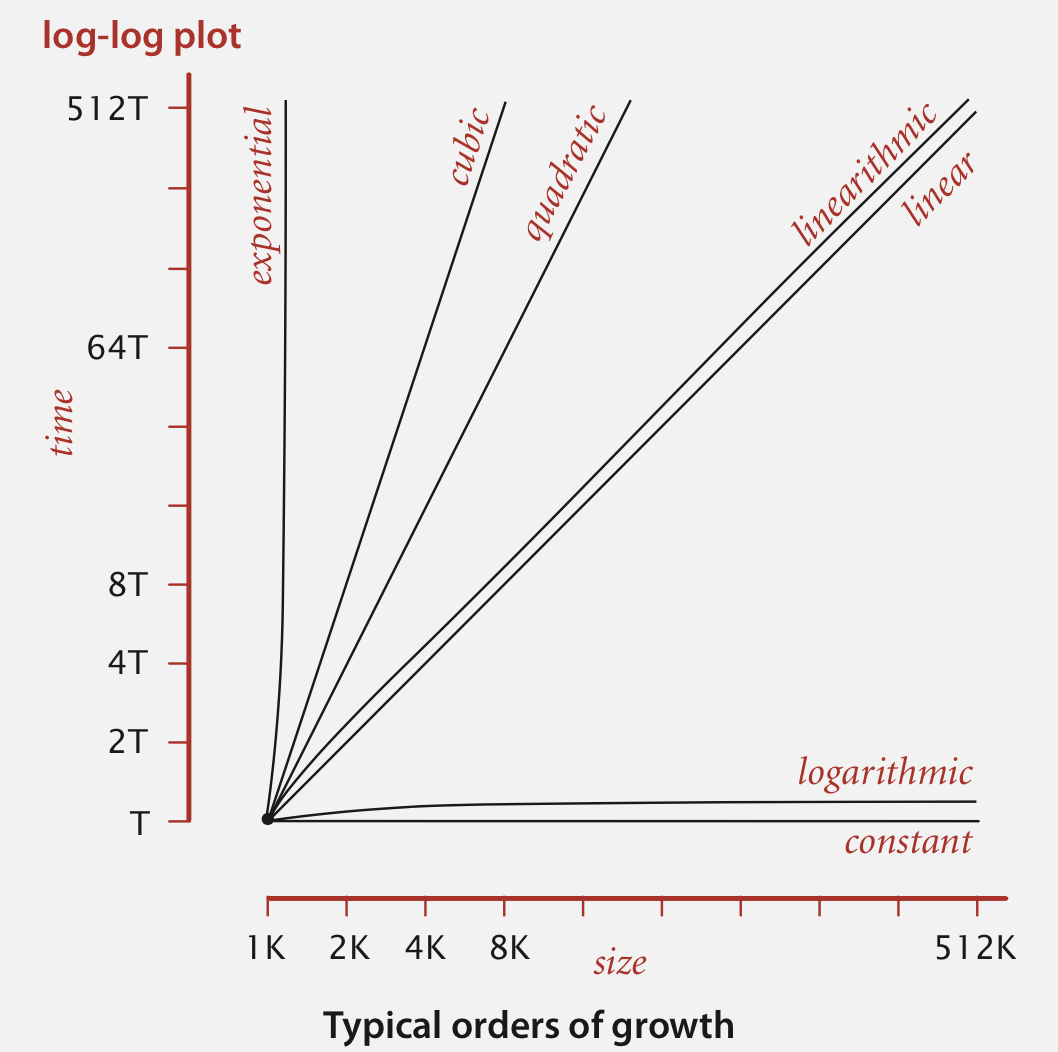
}

i = i +1; c8 N

}

**Order-of-growth classifications:**

****



**Problems**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Algortihms/n | 100 | 300 | 500 | 700 | 1100 | 1300 | 1500 | 1700 | 1900 | 2100 | 2300 | 2500 |
| Matrix Multiplication | 86 | 859,8 | 4629,3 | 13044,6 | 53964,3 | 95357,6 | 143887,6 | 230578 | 357410,3 | 459179 | 593139,5 | 734587,5 |
| Bubble Sort | 3,8 | 17,6 | 26 | 27,2 | 33,2 | 34,8 | 29,2 | 36,2 | 37,6 | 47,6 | 55,4 | 56,8 |
| Finding Max | 2,0 | 2,6 | 2,8 | 3,6 | 4 | 4,4 | 5,2 | 5,8 | 6,4 | 7 | 7,8 | 8,4 |
| Merge Sort | 6,2 | 9 | 15,6 | 18 | 25,2 | 28 | 28,2 | 31 | 34,4 | 37,6 | 37,8 | 43,6 |
| Binary Search | 5,6 | 16,2 | 25,6 | 26,4 | 34,2 | 34,4 | 34,4 | 38,8 | 44,6 | 51 | 57,6 | 66,4 |

1. **Matrix Multiplaction:**

In this problem, it is expected from us multiply two nxn square matrices. It is pseudocode for the multiplaction:

for i = 1 to N

for j = 1 to N

c (i, j) = 0

for k = 1 to N

c(i, j) = c (i, j) + a(i, k) \* b(k, j)

end

end

end

As shown there are three *for loops* from 1 to N. Therefore, the time complexity is N3.

In my design I write a method which is creating nxn matrix and fill in with randomly integer numbers from 0 to 2500. This method has two nested *for loops*. Time complexity for this method is N2.

As a conclude, based on the last step of given instructions before, I add N3 and N2. N2 is trivial when it is compare with N3 so time complexity of matrix multiplaction is approximately N3(~N3).

Total Cost = c1\*1+c2\*1+c3\*1+c4\*N+c5\*N\*N+c6N2+c7\*N\*N\*N+c8\*N3 = ~N3 *(cubic)*

Numbers of input

1. **Finding Maximum Element**

It is expection from us to find maximum item from given array which is contains randomly integer numbers. It is pseudocode for this problem below:

func MaxElement(a)

max = 0;

for i = 0 to N

max = compare and return the bigger one (a[i], max)

return max

end func

Besides MaxElement(int[] a) method, I implement a makeArray(int n) method which is create an array and fillin it random numbers. It contains one *for loop*. It takes N time accessing array.

In MaxElement(int[] a) function, there is one *for loop* from 1 to N and program access N time the array.

Using the mathemetical analysis:

Total Cost = c1\*1+c2\*1+c3\*N+c4\*N = ~N *(linear)*

So, time complexity of the problem finding maximum element has linear growth. We can see this graph which is given below.

Numbers of input

1. **Bubble Sort**

In this part, it is expected from us to implement a bubble sort algorithm. I use pseudocode which is given in the assigment paper:

Bubblesort(var a as array)

for i from 1 to N

swaps = 0

for j from 0 t o N-i

if a[j] > a[j+1]

swap (a[ j ] , a[ j + 1])

swaps = swaps + 1

if swaps = 0

break

end func

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position.

As I did in the other parts of the experiment, I implemented a method which is make an array and fill with random numbers *makeArray(int n).* Its time complexity is N because of one for loop.

Bubble Sort method has two nested for loops from 1 to N and from 1 to N-i. It means we access to array N2 times.

In mathematical formula;

Total Cost = c1\*1+c2\*N+c3\*N+c4\*N\*N+c5\*N2+c6\*N2+c7\*N2+c8\*N2+c9\*N2=~N2

Bubble sort’s best, worst and average cases are equal to O(N2) and it has quadric graph. But in my test, I have found a different graph than what is expected. The reason for this is time complexity does not only affected by algorithm. As I mentioned before, there are a lot of factors which affects execution time. This is why I couldn’t get the expected graph in my work.

Numbers of input

1. **Merge Sort**

I work on this pseudocode to implement merge sort algorithm.

func mergesort (var a as array)

if ( n == 1 ) return a

var l1 as array= a [ 0 ] . . . a [ n / 2 ]

var l2 as array = a [ n / 2+1 ] . . . a [ n ]

l1 = mergesort ( l1 )

l2 = mergesort ( l2 )

return merge ( l1,l2 )

end func

func merge (var a as array , var b as array)

var c as array

while ( a and b have elements )

i f ( a [ 0 ] > b [ 0 ] )

add b [ 0 ] to the end of c

remove b [ 0 ] from b

else

add a [ 0 ] to the end of c

remove a [ 0 ] from a

while ( a has elements )

add a [ 0 ] to the end of c

remove a [ 0 ] from a

while ( b has elements )

add b [ 0 ] to the end of c

remove b [ 0 ] from b

return c

end func

In computer science, merge sort (also commonly spelled mergesort) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945. A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and Neumann as early as 1948.

Merge sort work like this. First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.

Merge sort’s time complexity is NlogN. It has linearithmic growth.

Numbers of input

1. **Binary Search**

The last part of the my experiment is that implement binary search algorithm. For

that use this pseudocode.

func BinarySearch (a, value , left , right)

while left <= right

mid = floor ((right-left)/2) + left

if a[mid] == value

return mid

if value < a[mid]

right = mid-1

else

left = mid+1

return not\_found

end func

In computer science, a binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomic divide-and-conquer search algorithm and executes in logarithmic time. Its time complexity is logN.

The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished. If the target value is less than the middle element's value, then the search continues on the lower half of the array; or if the target value is greater than the middle element's value, then the search continues on the upper half of the array. This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

Numbers of input

**Bibliography**

<http://www.mkyong.com/java/how-do-calculate-elapsed-execute-time-in-java/>

<http://introcs.cs.princeton.edu/java/22library/Matrix.java.html>

<http://aofa.cs.princeton.edu/10analysis/>

<http://introcs.cs.princeton.edu/java/41analysis/>

<https://en.wikipedia.org/wiki/Binary_search_algorithm>

<https://en.wikipedia.org/wiki/Merge_sort>

<https://en.wikipedia.org/wiki/Bubble_sort>