HACETTEPE UNIVERSITY

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

NAME-SURNAME: NEHİL DANIŞ

STUDENT ID: 21327876

SUBJECT: ANALYSIS OF ALGORITHMS

EXPERIMENT: 1/FINAL REPORT

PROGRAMMING LANGUAGE: JAVA

INTRODUCTION

As a computer engineering student , I always come up with problems.Lots of them have more than one solution.They are all correct.But not efficient and not suitable from every aspects.To call a solution as an efficient and a proper way , it has to be quick and at the same time it should take up less space than the other solution.In that time ,we meet some approachs to create the more efficient and take up less space algorithm.To acquire this ideas,we have to consider time and space complexity.

Running time of an algorithm depends on lots of factors.These are computer's qualities and also the input that we are giving to our program. We have to find the fastest way to solve the problem depends on these factors..Sometimes the running time difference can not be seen accurately.But when the input size gets bigger,the difference would be more visible.Correctness of the program is not the only thing that is important.The behaver of program for larger input sizes is also important.

When we talked about the time complexity analysis of an algorithm we have to consider input size more than the capabilities of our computer. We try to find the rate of growth of time.Some of expressions are taken smaller time ,because of that reason ,these are called as constant time taken algorithms.

```
Sum(a,b){
        return a+b;
}
```

One unit for return and one unit for addition.So these algorithm takes 2 units time.We can call it as a constant.It can be showed by using letter c.

To classify the running time of algorithm , we use asymptotic notations.There are there types of asymptotic notation.These are big-oh notation,theta notation and omega notation.

1.Big-Oh Notation: This always shows the upper bound of the rate of growth of time.Time cannot grow faster than this.

$f(n)=3n^2 + 5n + 1$

$f(n)\leq(3 + 5 + 1)\ n^2$, $n>0$

$f(n)=O(n^2)$

The running time of f(n) will be smaller than $9\ n^2$ depending on the big-Oh notation.

2.Omega Notation: This always shows the lower bound of the rate of growth of time.The running time of algorithm at least the measure of the omega notation.

$f(n) = 5n^2 + 3n + 1$

$5n^2$ grows faster than the other variables that stands on the f function.So the variables excepts $5n^2$ will be neglected.The lower bound of running time will be $5n^2$ .This means that ;

$f(n) \geq 5n^2$, $n>0$

$\Omega(n^2)$

3.Theta Notation: This is the best notation.Because this gives up the tight bound of the rate of growth of time.

$f(n) = 5n^2 + 2n + 1$

$c1g(n) \leq f(n) \leq c2g(n)$ c1 and c2 are constants.

C1=5 we can say that it shows the lower bound's constant.

C2=8 we can say that it shows the upper bound's constant.

$\Theta(n^2)$ This will shows the tight bound.So this is better than the other ones.

At the time complexity analysis,we usually try to find the tight bound of time taken.Because it gives us the best idea of time complexity.But we also use the big-oh notation a lot.It gives us an information of the worst case of an algorithm.

Apart from the time copmlexity , the space complexity is also important.Space complexity is the sum of number of cells that occupy.The smaller space complexity is the better.We have to solve problems by contolling both space and time complexity.The smaller space and time complexity is the best for the convenient solution.

ALGORITHMS

All the algorithms were tested between five and ten times for each size of inputs.I assigned the numbers randomly between 0 and 10000.Only in binary search algorithm ,to see the results clearly I assigned numbers randomly from 0 to n.

Matrix multiplication and bubble sort took much more time than the others.So we observed the result by using miliseconds.However,merge sort , binary search and finding max. element algorithms took less time than the other.So we will used nanosecond to observe results easily.

I analyzed algorithm except binary search and merge sort by using total cost. But in binary search and merge sort algorithm , there are some recusive calls.So it's meaningless to show it by using totaly cost.Instead of this one ,I used mathematical analysis.

Input Size	Matrix Multiplication(ms	Finding Max. Element(ns	Binary Search (ns)	Bubble Sort (ms)	Merge Sort (ns)
100	42,8	7186	156208	0,5	237134,6
300	146,6	18948,4	181438,2	6,1	784546,8
500	617,1	30668,4	184438,2	13	1372422,2
700	1809,4	42488,5	189571	15,9	1995970,4
1100	19111,8	71003,3	188459	19	3020764,4
1300	32836,2	83792,9	194532,8	20	3907678
1500	58629,2	89125,1	196447,4	22,5	4663681,4
1700	81250,4	133794,8	203172,8	25	5956260
1900	128956,6	151973,1	203689,2	26,6	7244872,2
2100	192813	161169,6	208444,6	27,4	8726426,6
2300	257763,5	175626,8	209160,8	31,9	10139312,6
2500	299487,5	193420,8	210647,6	35,5	11797631,4

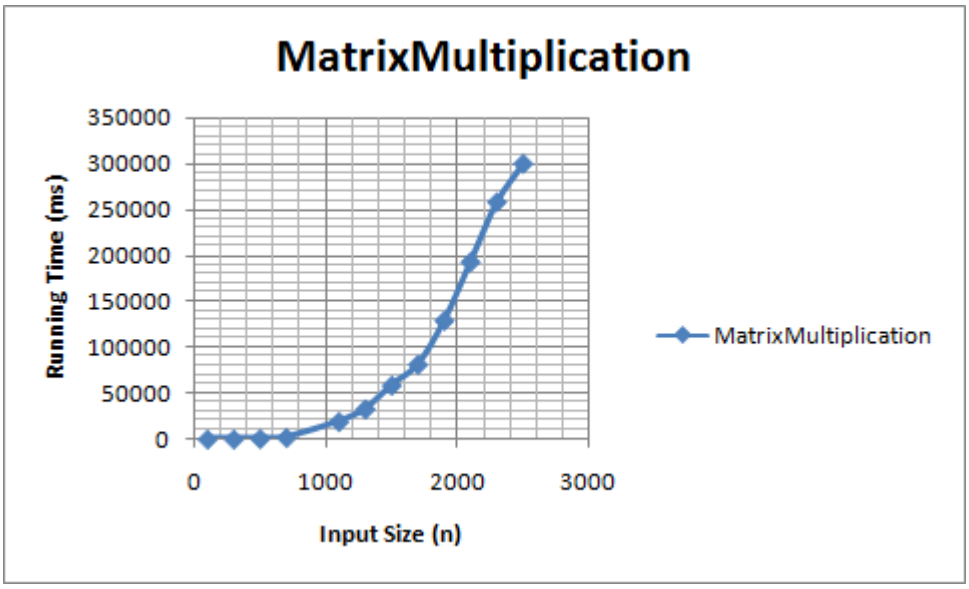
1. Matrix Multiplication Algorithm

	Unit Cost	Times
`for(int i=0;i<n;i++){`	c1	n+1
`  for(int j=0;j<n;j++){`	c2	n*(n+1)
`  c[i][j]=0;`	c3	n*n
`  for(int k=0;k<n;k++){`	c4	n*n*(n+1)
`  c[i][j]+=a[i][k]*b[k][j];`	c5	n*n*n
`  }`		
`  }`		
`}`		

Total Time Cost : (n+1)*c1+(n*(n+1))*c2+n*n*c3+(n*n*(n+1))*c4+n*n*n*c5

The time required for this algorithm is proportinal to n^3 which is determined as growth rate and it is usually denoted as $O(n^3)$

The extra space required for this algorithm is $4n^2 + 24$ which is denoted as $O(n^2)$



In matrix multiplication algorithm ,we use three for loop .First two are for assigning 0 to the result array.And we have to reach each element of result array to assign the result of multiplication.And the last for loop is for the multipliers.

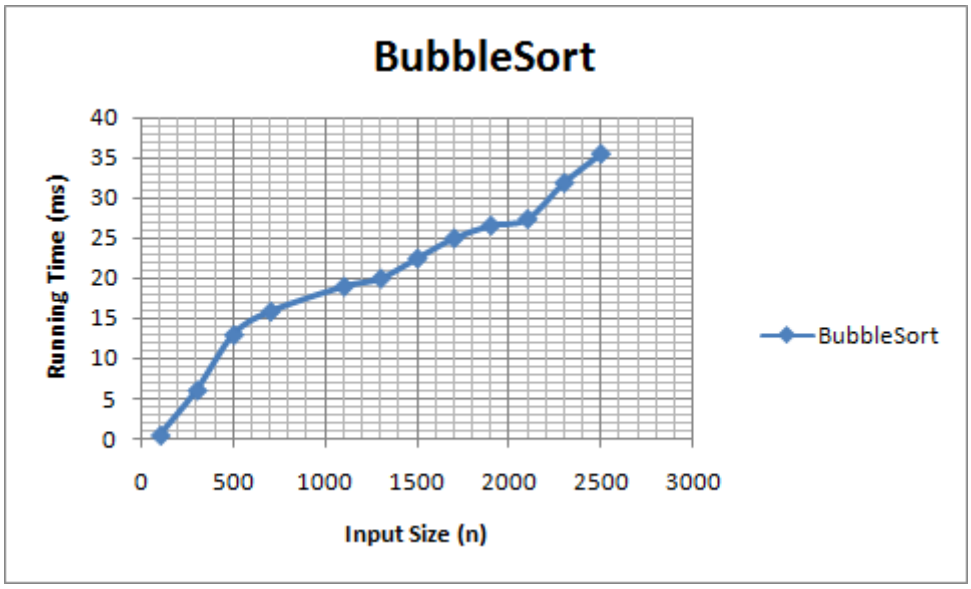
These for loops produces n^3 time complexity.And we are using the two dimensioned result array.This array occupy $4n^2 + 24$ which is denoted by $O(n^2)$. 24 bytes for array overhead in Java.n^2 comes from the number of dimensions.Each one has n element.This array is an integer array so we multiplied n^2 by 4 bytes.

2. Bubble Sort Algorithm

```
for(int i=1;i<array.length;i++){                    c1      n
   int swaps=0;                                     c2      n-1
   for(int j=0;j<(array.length-i);j++){             c3      (n-1)*(n-i+1)
      if(array[j]>array[j+1]){                      c4      (n-1)*(n-i)
            int var=array[j+1];                     c5      0 to (n-1)*(n-i)
            array[j+1]=array[j];                    c6      0 to (n-1)*(n-i)
            array[j]=var;                           c7      0 to (n-1)*(n-i)
            swaps+=1;                               c8      0 to (n-1)*(n-i)
      }
   }
   if(swaps==0){                                    c9      n-1
      break;                                        c10     0 to n-1
    }
}
```

Total Cost: n*c1 + (n-1)*c2 + (n-1)*(n-i+1)*c3 + (n-1)*(n-i)*c4 + (0 to (n-1)*(n-i))*c5 + (0 to (n-1)*(n-i))*c6 + (0 to (n-1)*(n-i))*c7 + (0 to (n-1)*(n-i))*c8 +(n-1)*c9 + (0 to (n-1))*c10

The time required for this algorithm is proportinal to n^2 which is determined as growth rate and it is usually denoted as $O(n^2)$
The extra space required for this algorithm is O(1).We just used a variable for counting swap times.This variable is integer ,so it occupy 4 bytes memory.This is denoted as O(1).



In this algorithm we have to for loops .One of them is for choosing a number from 1 to N .And then compare this number with the other numbers of array.For implementing this , we need second for loop .In this way we can compare the numbers and if it is necessary,we can swap elements.

There are some deviations in algorithm,this is because of distribution of best cases and worst cases and average cases for each input size.

3. Finding Maximum Element Algorithm

	Unit Cost	Times
`int max=0;`	c1	1
`for(int i=0;i<n;i++){`	c2	(n+1)
`  if(max<array[i]){`	c3	n
`  max=array[i];`	c4	0 to n
`  }`		
`}`		

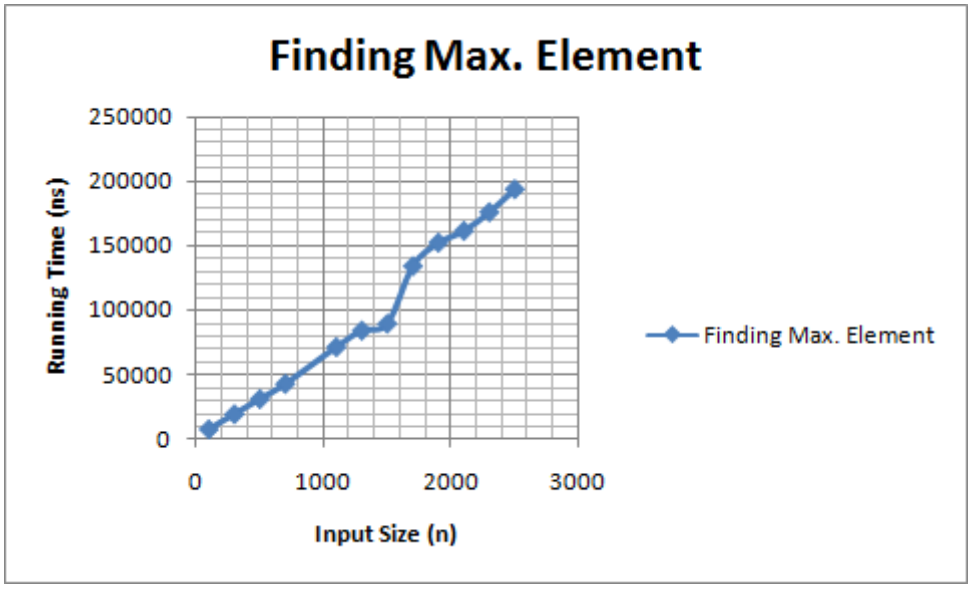
Total Time Cost : (n+1)*c1+(n*(n+1))*c2+n*n*c3+(n*n*(n+1))*c4+n*n*n*c5

The time required for this algorithm is proportinal to n which is determined as growth rate and it is usually denoted as O(n).

The extra space required for this algorithm is O(1).We use a max value in finding

maximum element algorithm.The value is occupy 4 bytes.Because it is an integer.The array is passed through as a parameter so we won't account the space required for array.So the extra space complexity will be shown by O (1). There is one for loop to take a look to the each element of array.This for loop gets n times.

The graph of this algorithm has to be linear.Because the time complexity is O(n).There are some deviations from original linear line graph.Reason of this is we decided the values of array randomly.We can come up with worse case ,best case ,and average case.So the changes will depend on the number of each these cases for each input size.



4. Merge Sort Algorithm

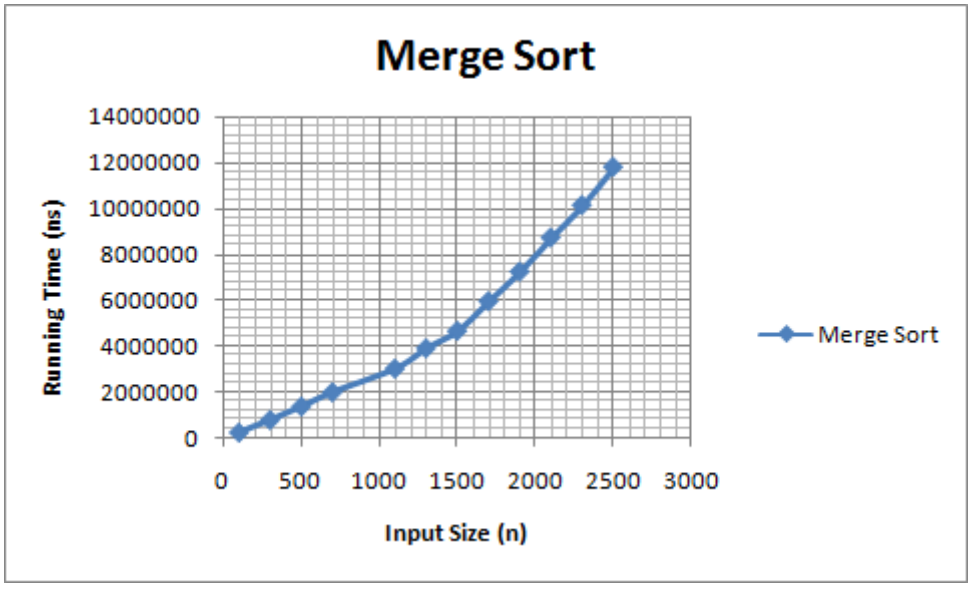
```
mergeSort(int [ ] a)

{

      int[] tmp = new int[a.length];
      mergeSort(a, tmp,  0,  a.length - 1);
}
mergeSort(int [ ] a, int [ ] tmp, int left, int right)
{
      if( left < right )
      {
            int center = (left + right) / 2;
            mergeSort(a, tmp, left, center);
            mergeSort(a, tmp, center + 1, right);
            merge(a, tmp, left, center + 1, right);
      }
}
```

```
merge(int[ ] a, int[ ] tmp, int left, int right, int rightEnd )
{
      int leftEnd = right - 1;
      int k = left;
      int num = rightEnd - left + 1;

     while(left <= leftEnd && right <= rightEnd){
           if(a[left]<=a[right])
               tmp[k++] = a[left++];
           else
               tmp[k++] = a[right++];
     }
        while(left <= leftEnd){
           tmp[k++] = a[left++];
     }
        while(right <= rightEnd){
           tmp[k++] = a[right++];
     }
        for(int i = 0; i < num; i++, rightEnd--){
           a[rightEnd] = tmp[rightEnd];
     }
}
```



The time required for this algorithm is proportinal to n*lgn which is determined as growth rate and it is usually denoted as O(n*lgn).

The extra space required for this algorithm is O(n).In merge sort algorithm we use a copy of orginal array.Because of that reason we use extra n sized array.This array is a one dimensional integer array.It occupy 4n+24 bytes. This can be denoted as O(n).

$T(N) = T(N/2) + T(N/2) + N$

These two T(N/2) are for sorting part of each half of an array.The N part is for merging these two sorted half each other. If the input size was 1, T(1) would be 0.

$T(1) = 0$
$T(N/2) = 2T(N/4) + N/2$

$T(N) = 2(2T(N/4) + N/2) + N$
$= 4T(N/4) + 2N$
$= 8T(N/8) + 3N$
. . .
. . .
. . .
$= 2^k T(N/2^k) + kN$

For $2^k = N$;
$k = \lg N$

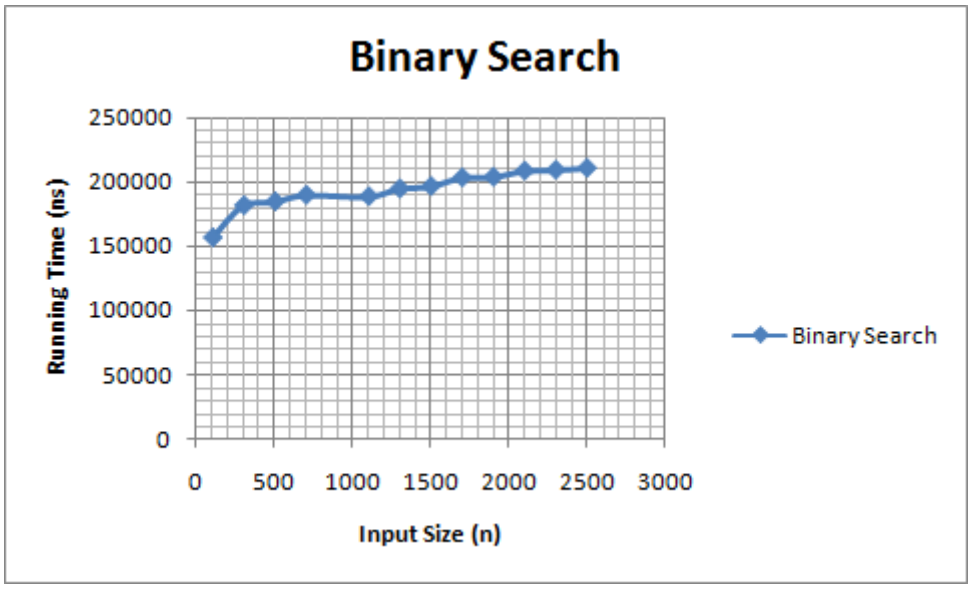
$=N\ T(N/N) + (\lg N)\ N$
$T(N/N) = T(1) = 0$

$= N \lg N$

5. Binary Search Algorithm

```
while(left<=right){
      int mid=(left+right)/2;
      if(a[mid]==value){
            return mid;
      }
      else if(a[mid]>value){
            right=mid-1;
      }
      else {
            left=mid+1;
      }
}
return -1;
```

The time required for this algorithm is proportinal to lgn which is determined as growth rate and it is usually denoted as O(lgn).
The extra space required for this algorithm is O(1).We just use two int variables for left and right.The array pass throgh the function as a parameter.So we don't have to consider it.These variables occupy 8 bytes.The extra space complexity is O(1), because there is any space that proportional to n .



There are some deviation on the graph different from the graph of lgn . I tested each input size 5 or more times.Some of the times , I met best case .Some of the others,I met average or worst case.These factors affected the testing result and also the graph.Sometimes I have met worst case for 3 times or more ,and this increase the running time.But sometimes,I have met best case more than the others, this decrease the running time.Depends on that reasons there are occured some deviation

When we get in the while loop,everytime we will calculate the middle variable.And the if we find the target ,the program will out of the while loop.Else ,each time we will take the half of array.So each step , number of loop will decrease the half of last time.

$$\sum_{i=1}^{n} \frac{n}{2\text{^}i} = n/2+n/4+n/8+\ldots+1=\lg n$$

T(N) = T(N/2) + 1

T(N/2) is for controlling the lef tor right half of array.1 is for implementing the convenient way. If the input size was 1, T(1) would be 0.

T(1) = 0

T(N) = T(N/2) +1
T(N) = (T(N/4) + 1) +1
=((T(N/8) + 1) + 1) + 1
. . . .
. . . .
. . . .
$=T(N/2^k) + k$
For $2^k = N$, k = lgN

= T(N/N) + lgN T(1) = 0 → **= lgN**