Analyse Different Sorting Algorithms Theoretically and Experimentally

جامعة الإمام عبد الرحمن بن فيصل
IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY
كلية العلوم والدراسات الإنسانية - علوم الحاسب
College of Science and Humanities - Computer Science

"CS 412 Algorithm Analysis and Design (Term 1, 2020-2021)"

Teamwork:
Nawal A. Alamri
Masomah Alshaer
Wasan Alshehri
Dana Rami Albandari Alsanhani
Supervised by:
Dr. Azza Ahmed Ali

Agenda

- Theoretical Questions
- Data Generation and Experimental Setup
- Which of the three sorts seems to perform the best?
- To what extent does the best, average, and worst-case analyses of each sort agree with the experimental results
- Comparison sorts
- Part 2

Introduction

-Insertion sort

-Heap sort

-Merge sort

Insertion sort

It looks as breaking the data into two subarrays. The right array is not sorted while the left array is sorted. Compare the first element in the left array with the elements in the right array then insert it into the correct place

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Insertion sort algorithm

	Best Case	Average Case	Worst Case
Time complexity	T(n)=O(n)	T(n)= O(n^2)	T(n)= O(n^2)

Heap sort

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       exchange A[1] with A[i]
4       A.heap-size = A.heap-size − 1
5       MAX-HEAPIFY(A, 1)
```

Heap sort algorithm

Time complexity	Best Case	Average Case	Worst Case
	T(n)=O(n lg n)	T(n)= O(n lg n)	T(n)= O(n lg n)

Merge sort

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
 4  for i ← 1 to n1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7        do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

Merge sort algorithm

	Best Case	Average Case	Worst Case
Time complexity	T(n)=O(n lg n)	T(n)= O(n lg n)	T(n)= O(n lg n)

Data Generation and Experimental Setup

Description of the characteristics of the machine used.

# Of PC / Characteristics	PC 1	PC 2
OS	Windows 11	MacOS Big Sur
Processor	Intel core i5	Apple M1 chip with 8-core CPU, and 16 core Neural Engine
System type	64-bit	64-bit

Time Mechanism.

This project used :: now() function, this function in C++ language returns the current time which is Microsecond

Did you use the same inputs for all sorting algorithms?

Yes, cause its fair enough to compare between all type of sorting algorithm and to get accurate result.

How many times did you repeat each experiment ?

We use input 10000,15000,20000,25000,320000 by adding 5000 five time. We use these number because it is big enough to get running time for each algorithm. We stop at value 320000 because bigger number its cause the program to lag

What times are reported?

We calculate the time by microseconds unit in each algorithm

What is the input to the algorithms? How did you select it?

The input was numbers, and we used a random sequence that generates using the rand() function

Selection of the best performance version of Merge sort?

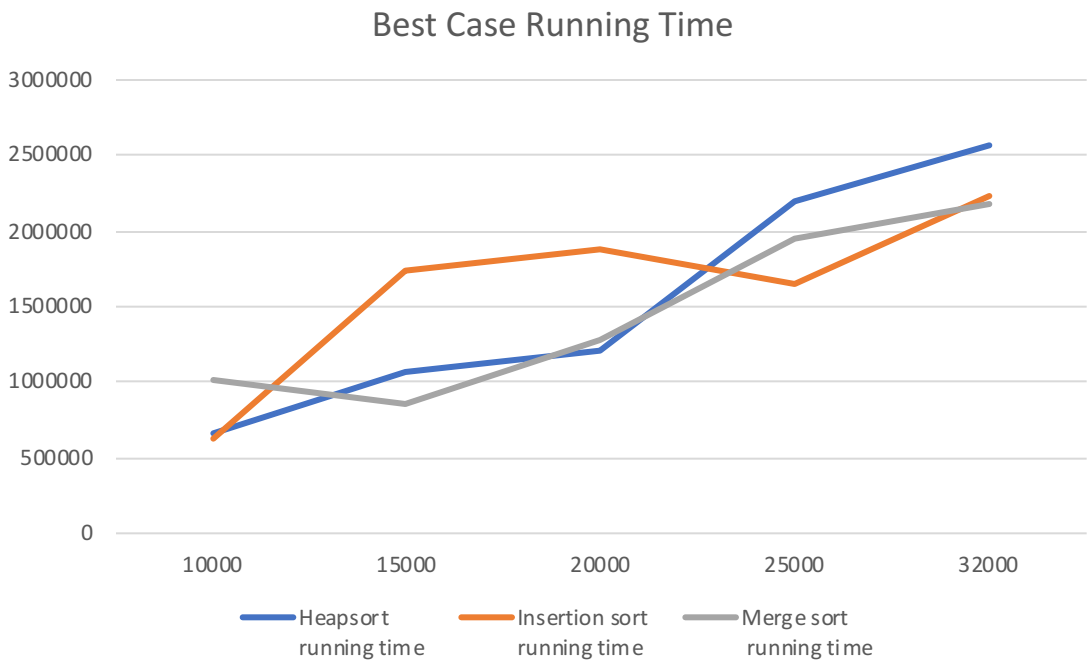
-Best case

-Worst case

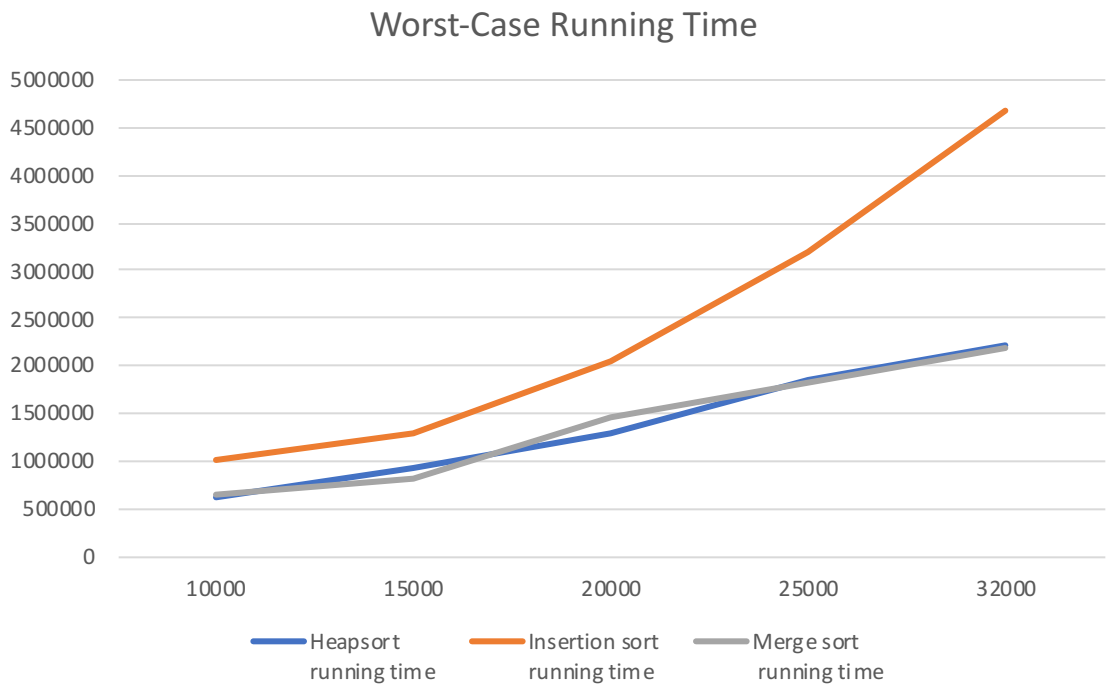
-Average case

-Discussion

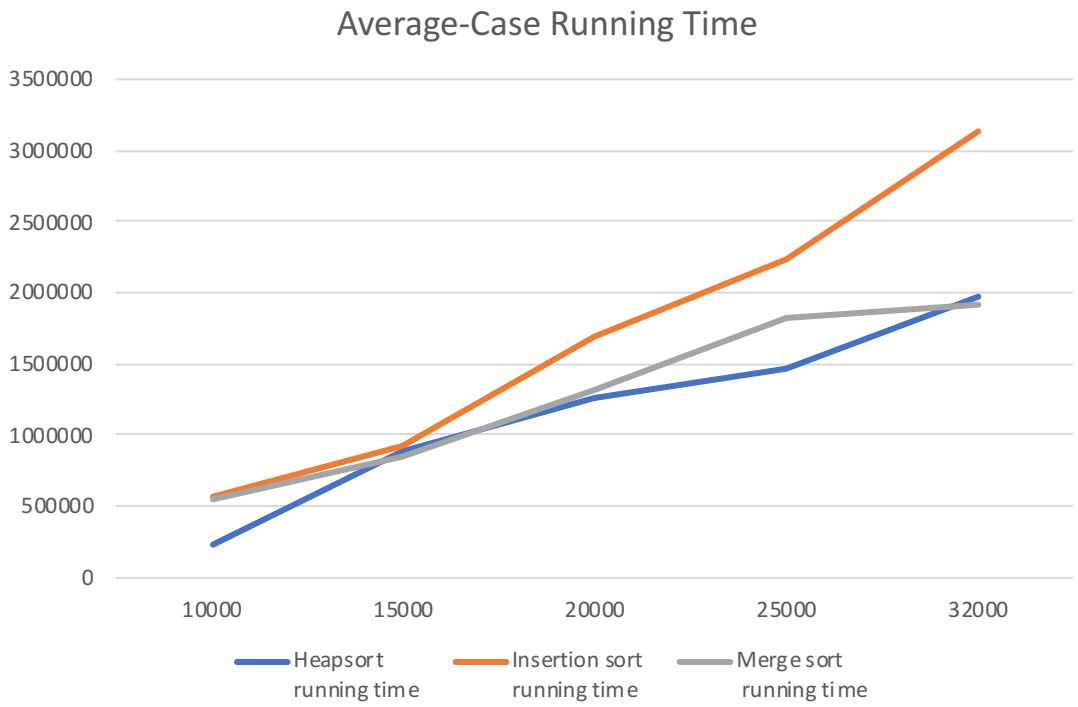
Graph the best-case running time as a function of input size n for the three sorts:



Graph the worst-case running time as a function of input size n for the three sorts:



Graph the average-case running time as a function of input size n for the three sorts:



Discussion of the best performance sorting algorithm:

According to the deep analysis of the sort algorithm from the figure it can be analyzed that the merge sort running time is considered as the best sort because it is distinguishable in all of the three different scenarios and the values considerably remains stable through the systematic approach applied. The three different scenarios present a whole different kind of nature for the parameters of the time and sorting. The best as per data records is the merge sorting

To what extent does the best, average and worst-case analyses of each sort agree with the experimental results?

-Insertion sort

-Merge sort

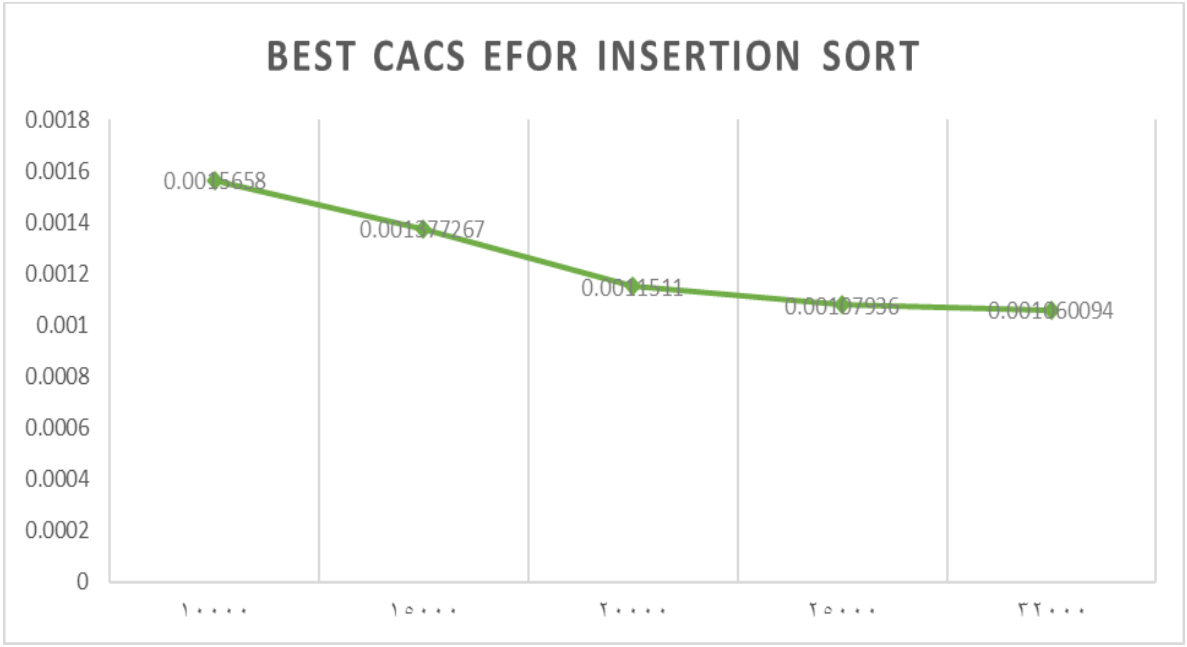
-Heap sort

Insertion sort

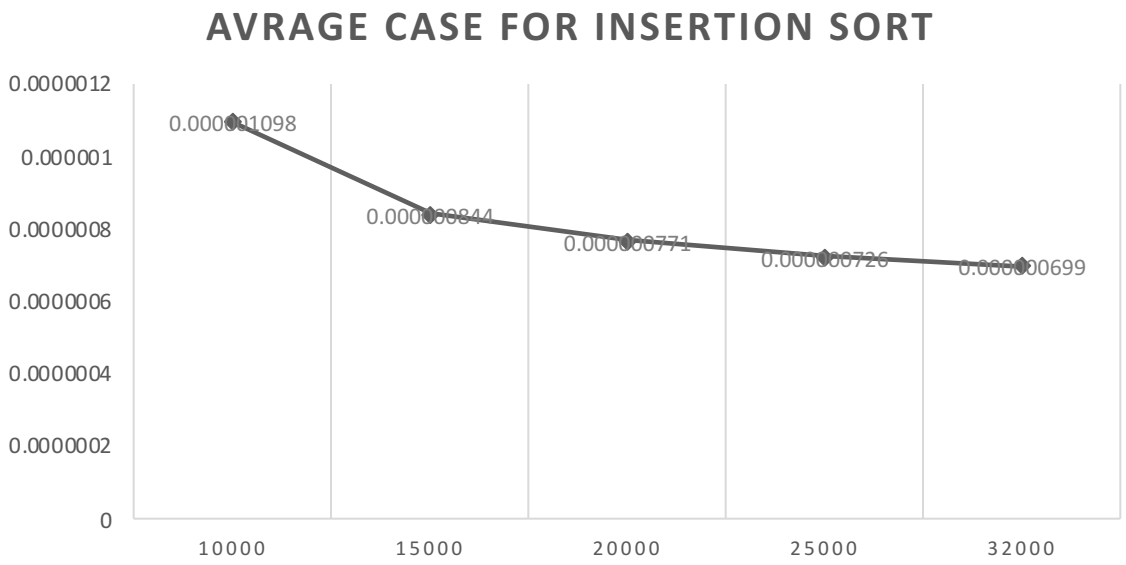
	Worst case (Decrease order)			Average case (Random order)			Bast cases (increase order)		
	T	P	P/T	T	P	P/T	T	P	P/T
10000	100,000,000 ms	148.086 ms	0.000001481 ms	100,000,000 ms	109.816 ms	0.000001098 ms	10000 ms	15.658 ms	0.001565800 ms
15000	225,000,000 ms	316.64 ms	0.000001407ms	225,000,000 ms	189.798 ms	0.000000844 ms	15000 ms	20.659 ms	0.001377267 ms
20000	400,000,000 ms	551.227 ms	0.000001378 ms	400,000,000 ms	308.331 ms	0.000000771 ms	20000 ms	23.022 ms	0.001151100 ms
25000	625,000,000 ms	850.774 ms	0.000001361 ms	625,000,000 ms	453.528 ms	0.000000726 ms	25000 ms	26.984 ms	0.001079360 ms
32000	1,024,000,000 ms	1381.194 ms	0.000001349 ms	1,024,000,000 ms	715.63 ms	0.000000699 ms	32000 ms	33.923 ms	0.001060094 ms

Insertion Sort Theory Vs. Experimental

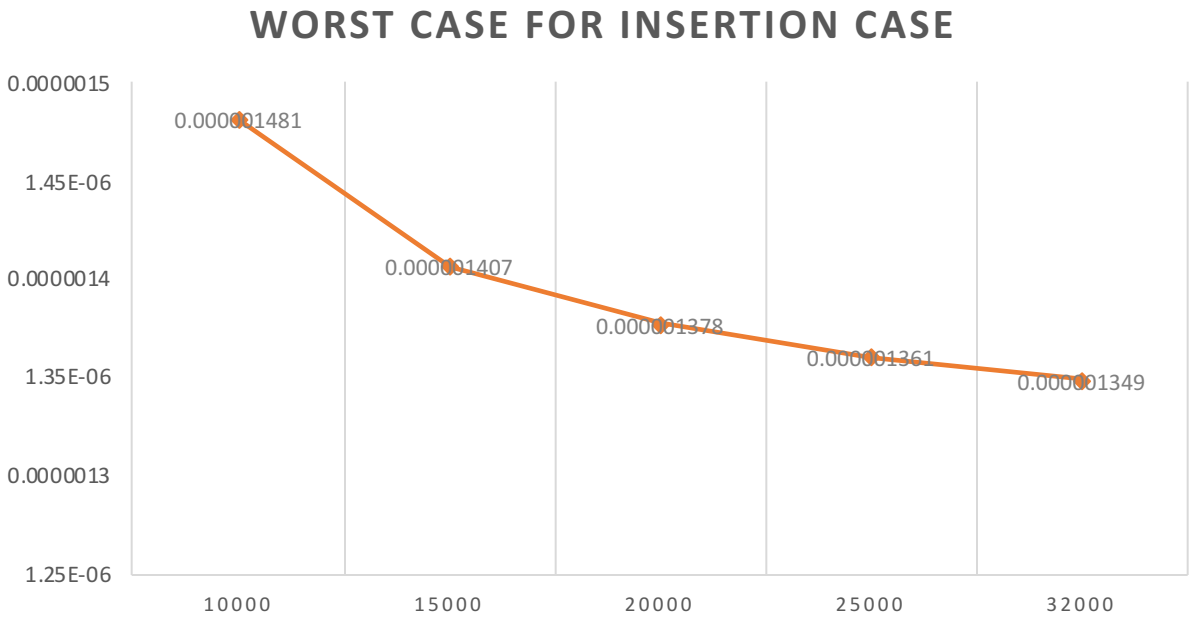
Insertion sort



Insertion sort



Insertion sort

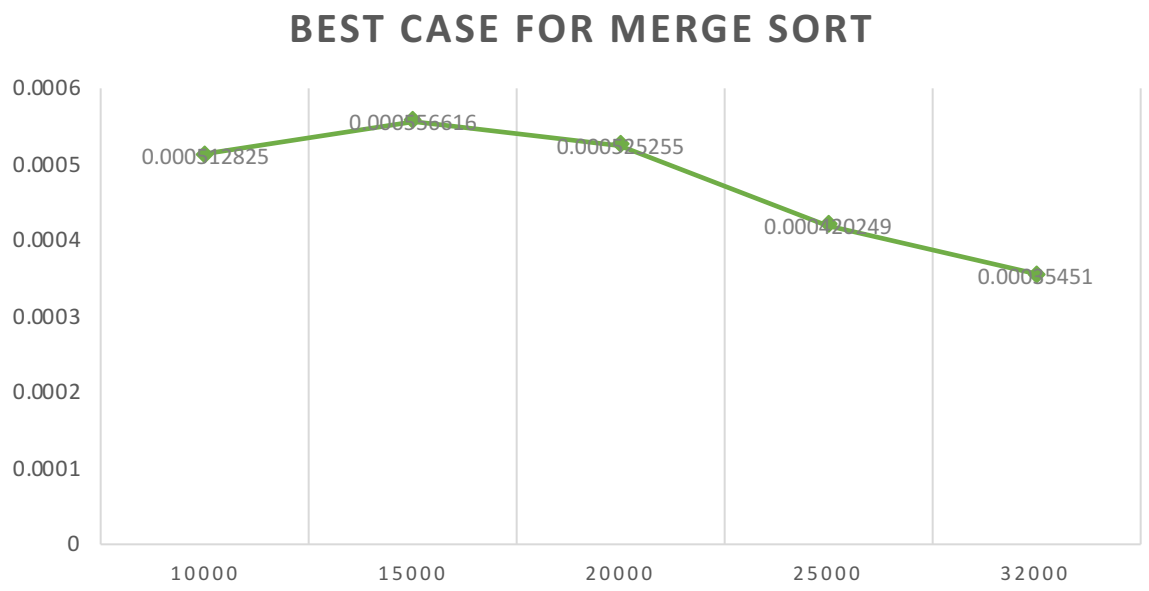


Merge sort

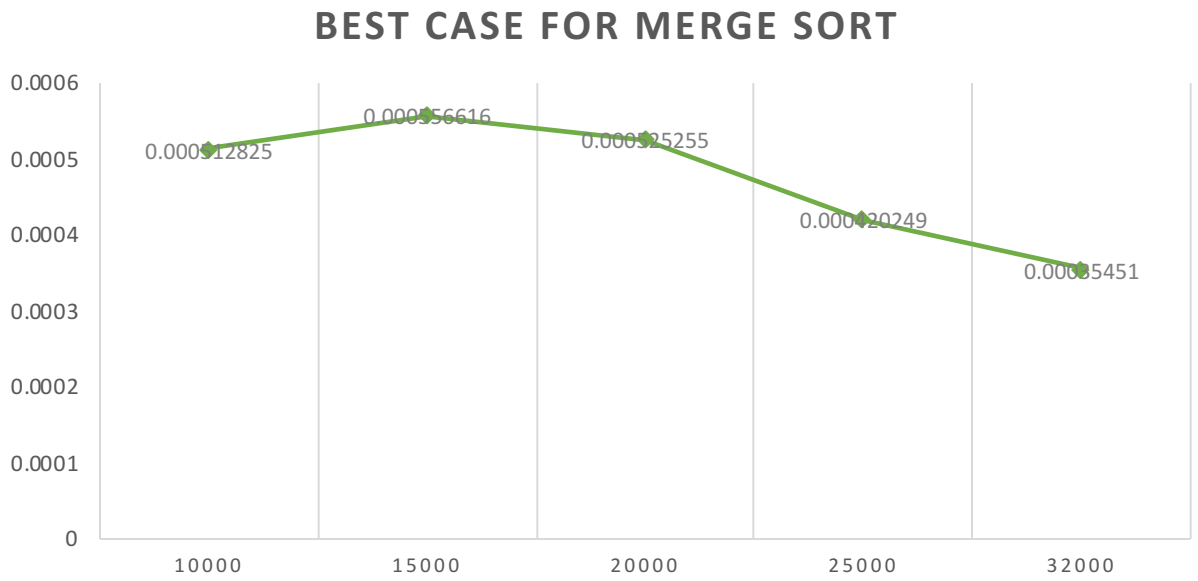
	Worst case (Decrease order)			Average case (Random order)			Bast cases (increase order)		
	T	P	P/T	T	P	P/T	T	P	P/T
10000	40000 ms	25.473ms	0.000636825ms	40000 ms	33.031 ms	0.000825775 ms	40000 ms	20.513 ms	0.000512825 ms
15000	62641 ms	33.441ms	0.000533852ms	62641 ms	34.435 ms	0.000549720 ms	62641 ms	34.867 ms	0.000556616 ms
20000	86021 ms	40.562ms	0.000471536ms	86021 ms	46.638 ms	0.000542170 ms	86021 ms	45.183 ms	0.000525255 ms
25000	109949 ms	48.462ms	0.000440768ms	109949 ms	55.743 ms	0.000506990 ms	109949 ms	46.206 ms	0.000420249 ms
32000	144165 ms	52.85ms	0.000366594ms	144165 ms	46.333 ms	0.000321389 ms	144165 ms	51.108 ms	0.000354510 ms

Merge Sort Theory Vs. Experimental

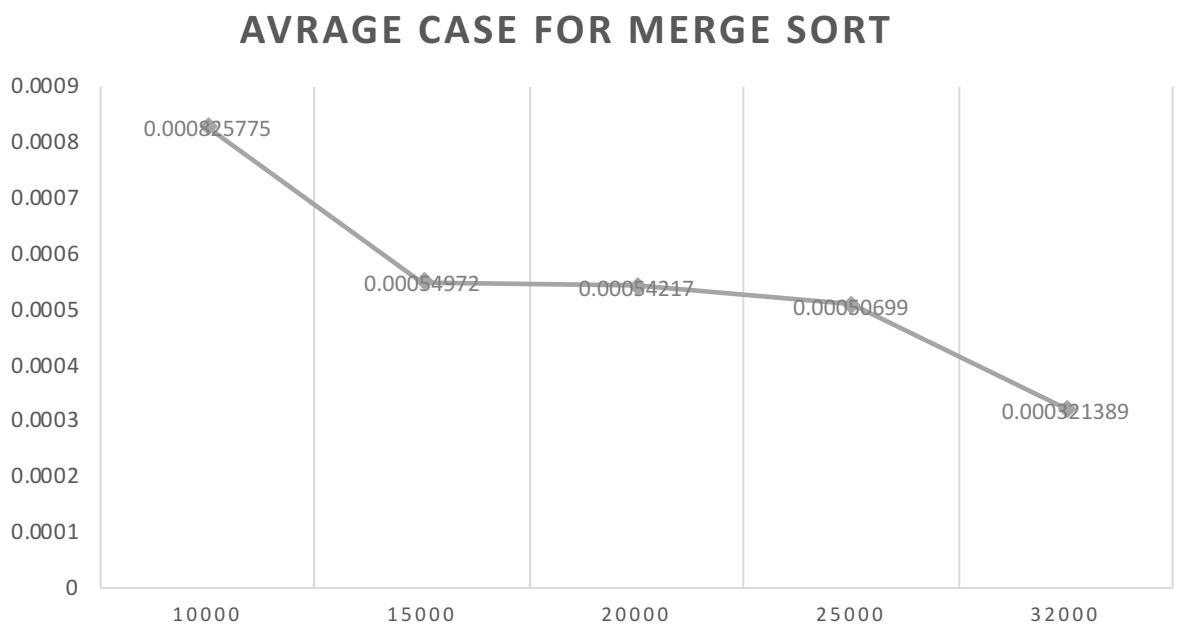
Merge sort


BEST CASE FOR MERGE SORT
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
10000
15000
20000
25000
32000

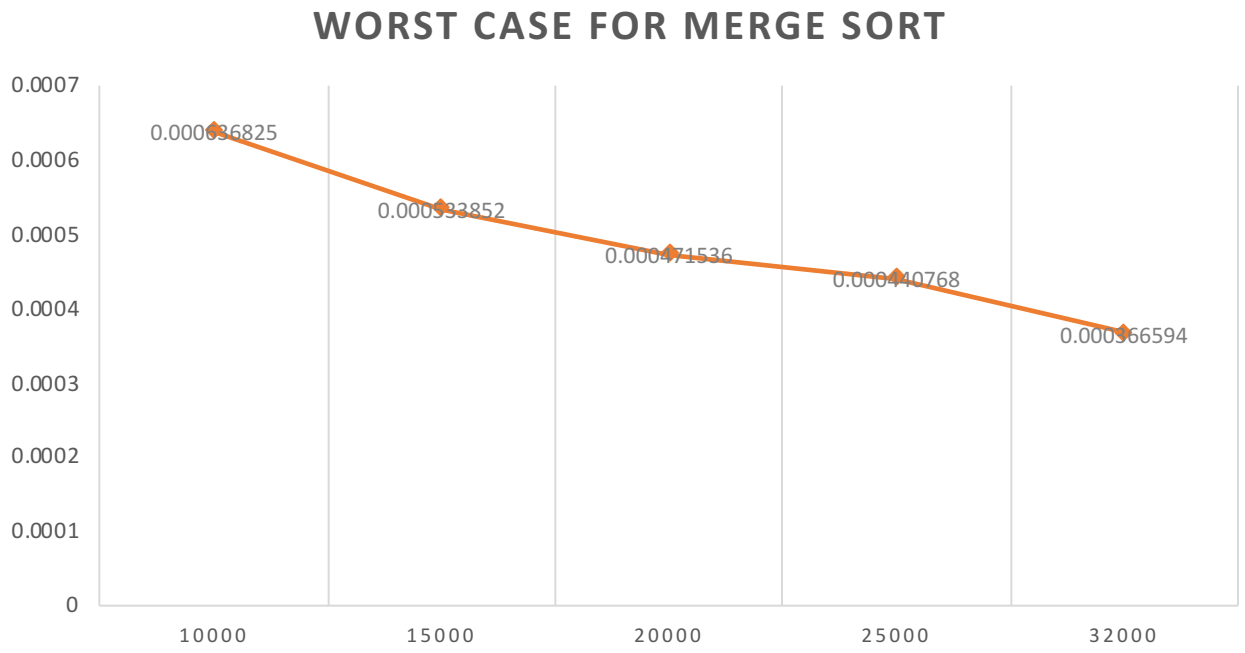
Merge sort


BEST CASE FOR MERGE SORT
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
10000
15000
20000
25000
32000

Merge sort


AVRAGE CASE FOR MERGE SORT
0.0009
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
0.000825775
0.000554972
0.000554217
0.000550699
0.000321389
10000
15000
20000
25000
32000

Merge sort

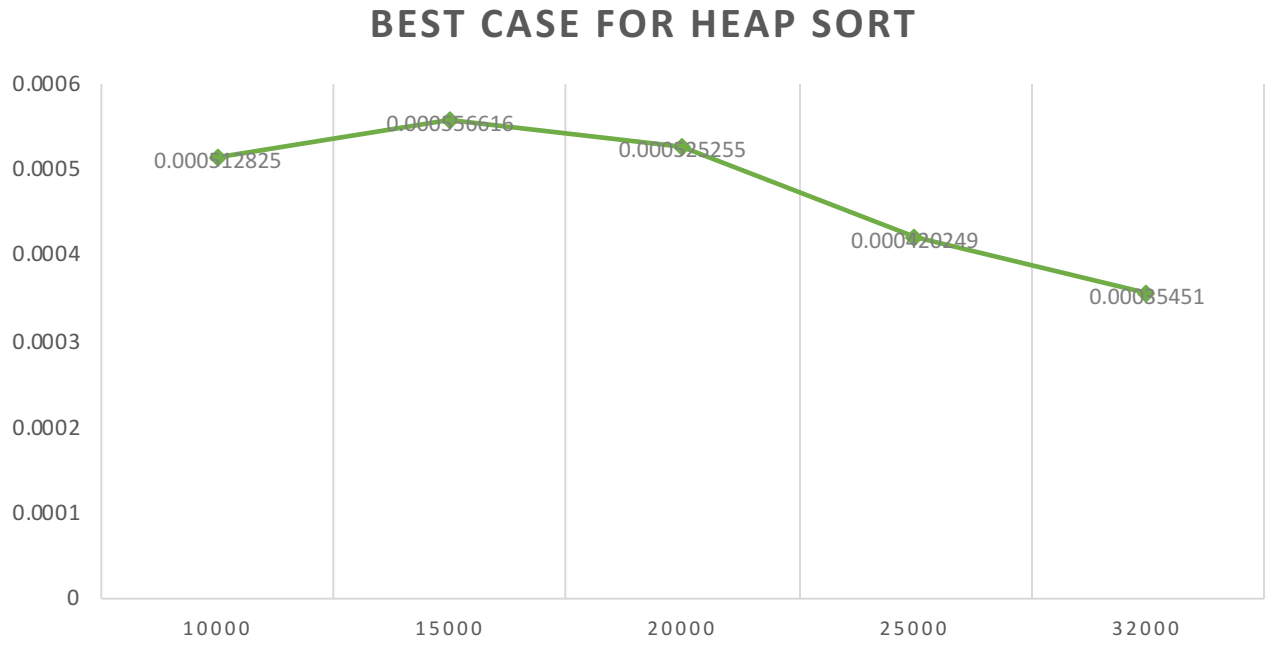

WORST CASE FOR MERGE SORT
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
0.000686825
0.000533852
0.000471536
0.000440768
0.000366594
10000
15000
20000
25000
32000

Heap sort

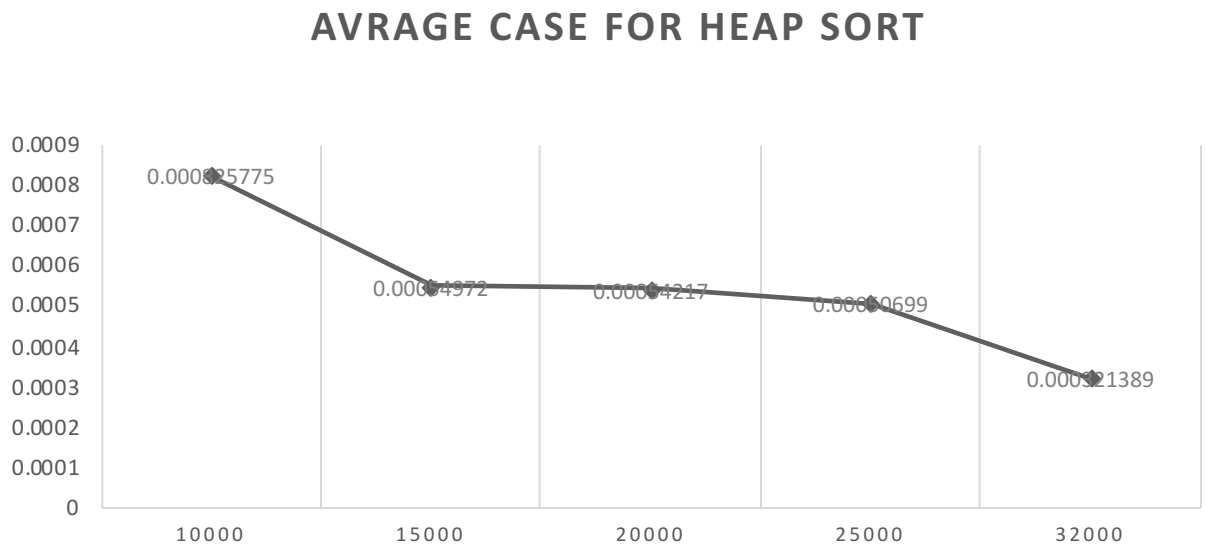
	Worst case (Decrease order)			Average case (Random order)			Bast cases (increase order)		
	T	P	P/T	T	P	P/T	T	P	P/T
10000	40000 ms	27.72 ms	0.000693000 ms	40000 ms	10 ms	0.000250000 ms	40000 ms	26.924 ms	0.000673 100 ms
15000	62641 ms	33.633 ms	0.000536917 ms	62641 ms	10.9 ms	0.000174007 ms	62641 ms	33.366 ms	0.000532 654 ms
20000	86021 ms	41.99 ms	0.000488137 ms	86021 ms	11.306 ms	0.000131433 ms	86021 ms	41.32 ms	0.000480 348 ms
25000	109949 ms	45.568 ms	0.000414447 ms	109949 ms	13.487 ms	0.000122666 ms	10994 9 ms	42.675 ms	0.000388 134 ms
32000	144165 ms	53.478 ms	0.000370950 ms	144165 ms	17.948 ms	0.000124496 ms	14416 5 ms	50.105 ms	0.000347 553 ms

Heap Sort Theory Vs. Experimental

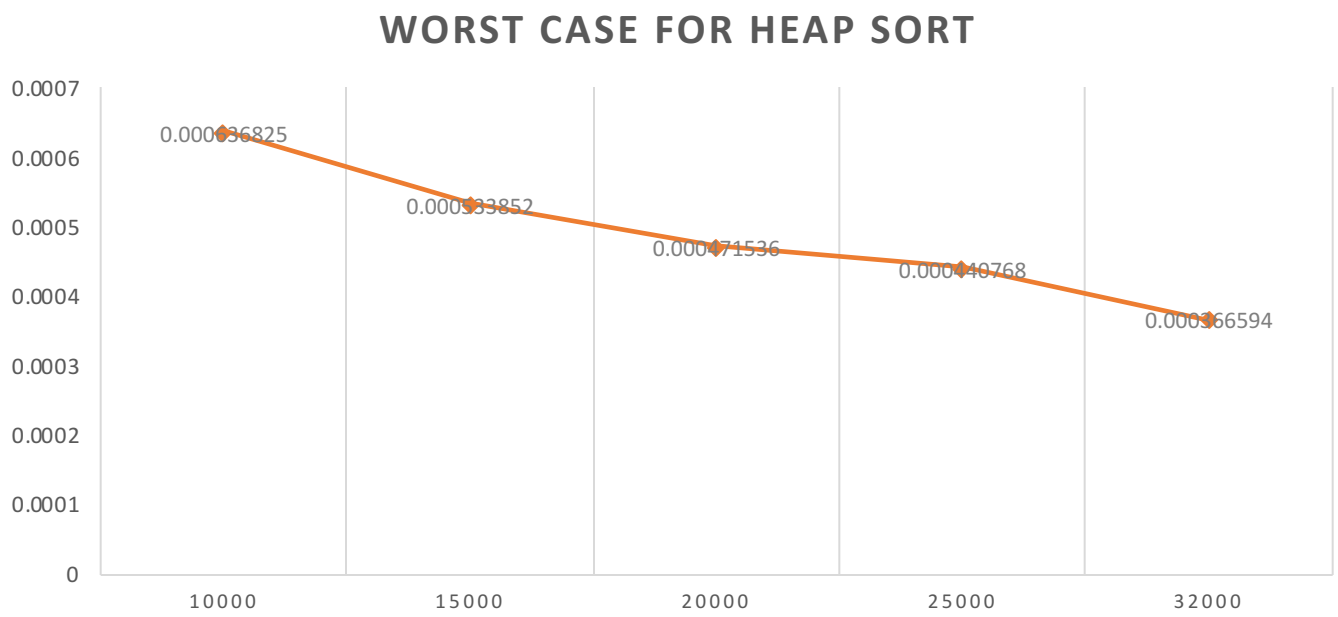
Heap sort


BEST CASE FOR HEAP SORT
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
0.000512825
0.000556616
0.000525255
0.000420249
0.00035451
10000
15000
20000
25000
32000

Heap sort


AVRAGE CASE FOR HEAP SORT
0.0009
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
10000
15000
20000
25000
32000

Heap sort


WORST CASE FOR HEAP SORT
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
10000
15000
20000
25000
32000

Comparison

-Insertion sort

-Heap sort

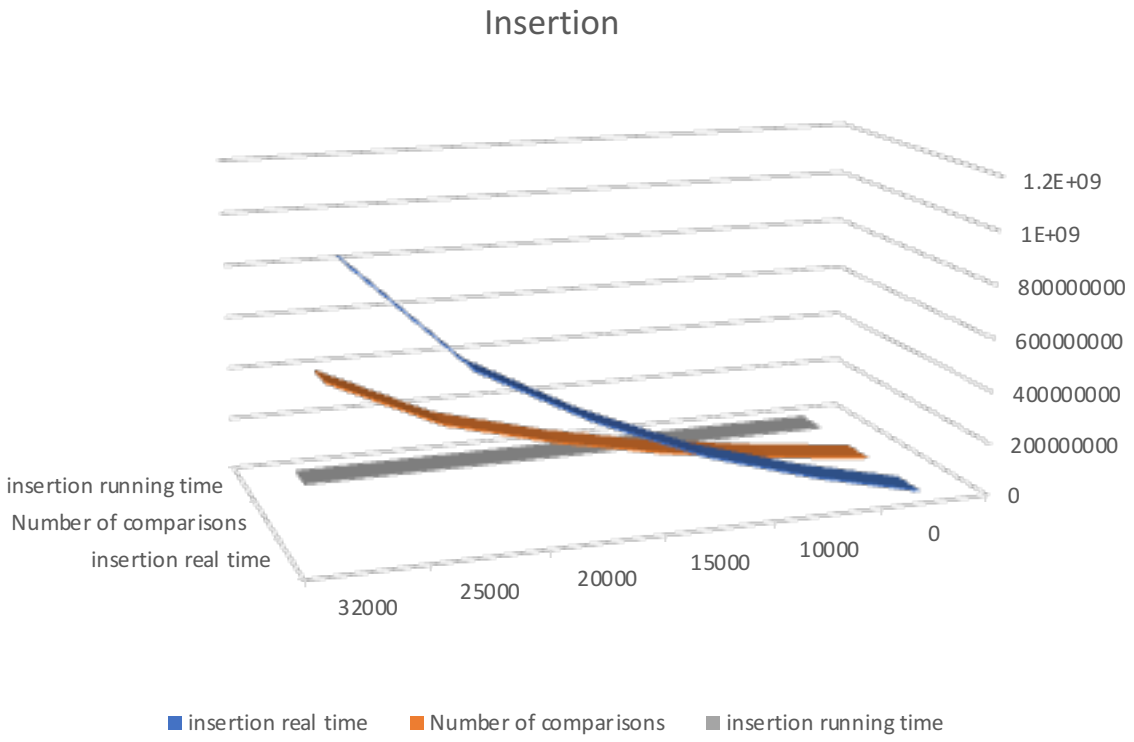
-Merge sort

Insertion sort

Number of elements	Insertion running time	Insertion real time	Number of Comparisons
10000	148.086	100,000,000	50000000
15000	316.64	225,000,000	112500000
20000	551.227	400,000,000	200000000
25000	850.774	625,000,000	312500000
32000	1381.194	1,024,000,000	512000000

Insertion Comparison

Insertion sort

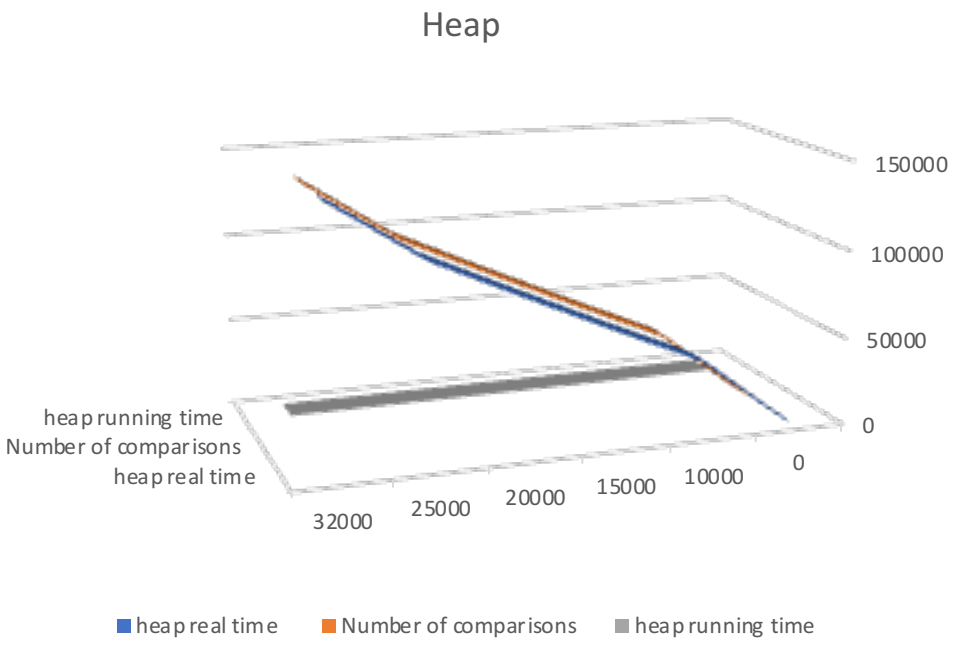


Heap sort

Number of elements	Heap running time	Heap real time	Number of Comparisons
10000	27.72	40000	40000
15000	33.633	62641	62641
20000	41.99	86021	86020
25000	45.568	109949	109948
32000	53.478	144165	144164

Heap Comparison

Heap sort

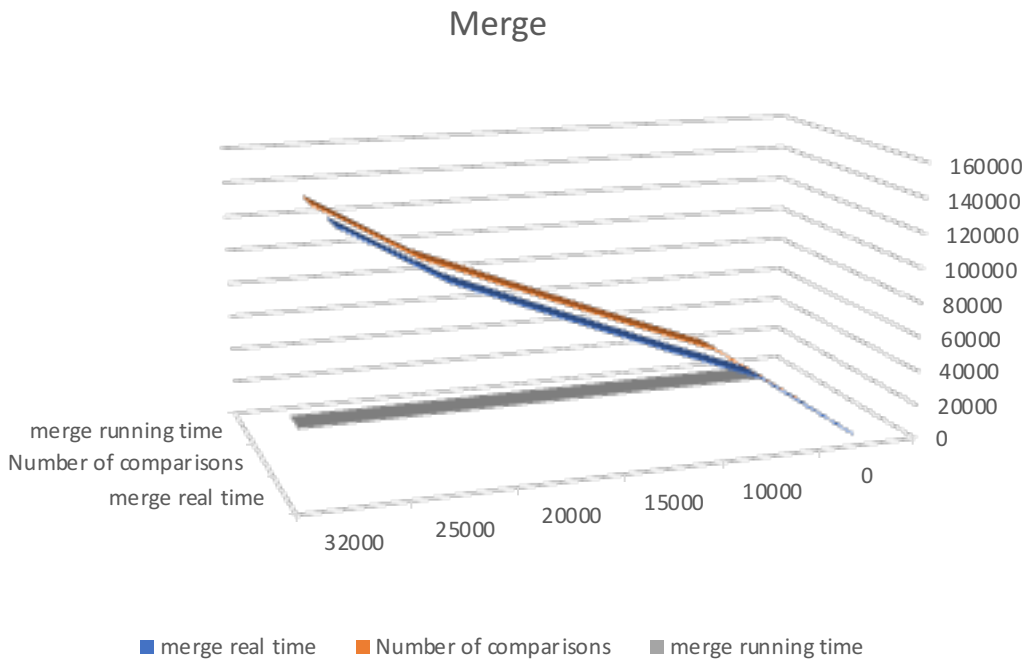

Heap
150000
100000
50000
0
heap running time
Number of comparisons
heap real time
32000
25000
20000
15000
10000
0
heap real time
Number of comparisons
heap running time

Merge sort

Number of elements	Merge running time	Merge real time	Number of Comparisons
10000	25.473	40000	40000
15000	33.441	62641	62641
20000	40.562	86021	86020
25000	48.462	109949	109948
32000	52.85	144165	144164

Merge Comparison

Merge sort


Merge
160000
140000
120000
100000
80000
60000
40000
20000
0
merge running time
Number of comparisons
merge real time
32000
25000
20000
15000
10000
0
merge real time
Number of comparisons
merge running time


Part 2

Design and analysis an improved Divide-and-conquer algorithm compute an, where n is natural number.

```
long power(int a, unsigned n)
{
    // if n^0 then 1
    if (n== 0)
        return 1;
    // if n is 0
    if (a == 0)
        return 0;
    if (n % 2 == 0) // n is even
        return power(a * a, n / 2);
    else
        return a * power(a * a, n / 2);
}
```

```
int main()
{

    int a;
    cout<<"Enter the base:";
    cin>>a;

    unsigned n;
    cout<<"Enter the exponent:";
    cin>>n;

    int pow = power(a, n);
    cout<<"The result="<<pow;

return 0;
}
```

Time Complexity : O(lg n)
Space Complexity: O(lg n) for recursive call

Design and analysis an improved Divide-and-conquer algorithm compute an, where n is natural number. Burt Force

```
1-Int base , exp ,i ,result =1;

2-cout<<“Enter base and
exponent”>;

3-cin >>base>>exp;

4-for( i=0; i< exp; i++)
{
 5- result=result *base;

}
6-cout<<“base <<“^”<<exp<<“=“<<
result;

7-return 0 ;
}
```

C1,1

C2,1

C3,1

C4,n

C5, n

C6,1

C7,1

Time Complexity : T(n)=T(n)+c =O(n)

Design and analysis an improved Divide-and-conquer algorithm compute an, where n is natural number.

Power-Divide-Conqure (a,n)	
1- if n==0	C1,1
2- return 1	C2,1
3-a==0	C3,1
4-return 0	C4,1
5-if n%2=0 //n is even	C5,1
6-power(a*a, a/2) //recursive	C6, T(n/2)
7-else //n is odd	C7,1
8- a*power(a*a , n/2)	C8, T(n/2)

Time Complexity : $T(n)=T(n/2)+c = O(\lg n)$
Space Complexity: $O(\lg n)$ for recursive call

Design and analysis an improved Divide-and-conquer algorithm compute an, where n is natural number.

```
Enter the base:2
Enter the exponent:0
The result=1
```

When exponent=0

```
Enter the base:0
Enter the exponent:2
The result=0
```

When base=0

```
Enter the base:2
Enter the exponent:2
The result=4
```

Discussion

Thank you for listening