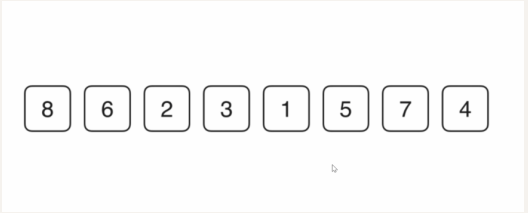
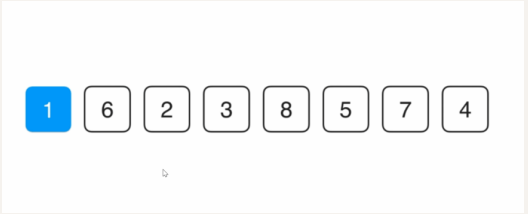
Time Complexities of Three Sorting Algorithms

CPS 3440 workshop 1

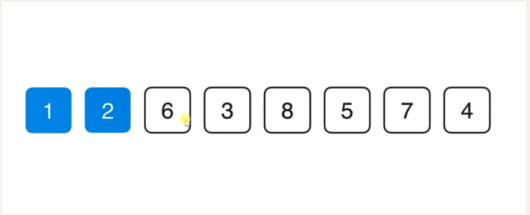
Selection Sort



First, select the smallest element



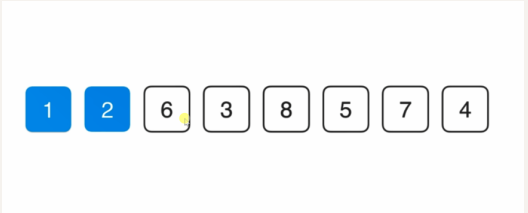
Then, select the smallest element among the remaining elements



Then, select the smallest element among the remaining elements

... ...

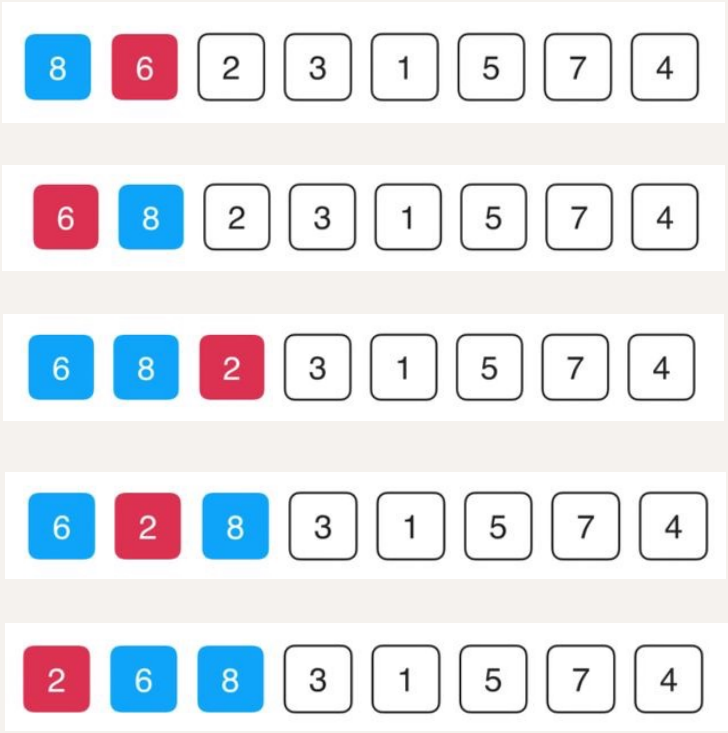
Select the smallest element among elements that have not yet been processed every time.



```java
public class SelectionSort{
    //  Our algorithm class does not allow any instances
    private SelectionSort(){}
    public static void sort(Comparable[] arr){
        int n = arr.length;
        for( int i = 0 ; i < n ; i ++ ){
            // Find the index of the smallest value in the interval [i, n)
            int minIndex = i;
            for( int j = i + 1 ; j < n ; j ++ )
                // Use compareTo method to compare the size of two Comparable objects
                if( arr[j].compareTo( arr[minIndex] ) < 0 )
                    minIndex = j;
            swap( arr , i , minIndex);
        }
    }
    private static void swap(Object[] arr, int i, int j) {
        Object t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
    }
}
```

Sorted: arr[0, i)
Out of order: arr[i, 0)

Insertion Sort



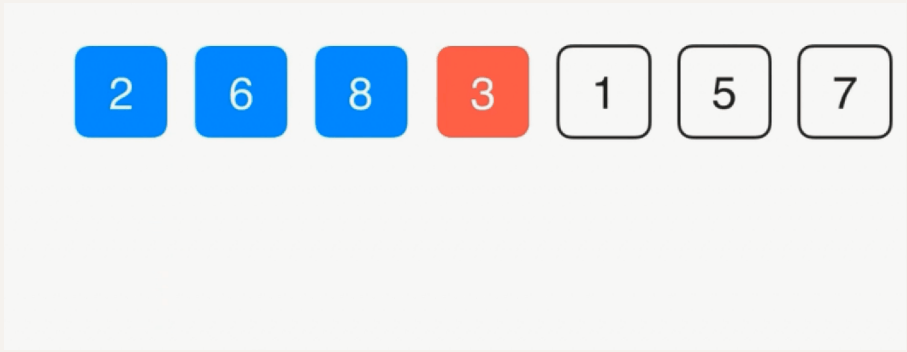
... ...

Sorted: arr[0, i)
Out of order: arr[i, 0)

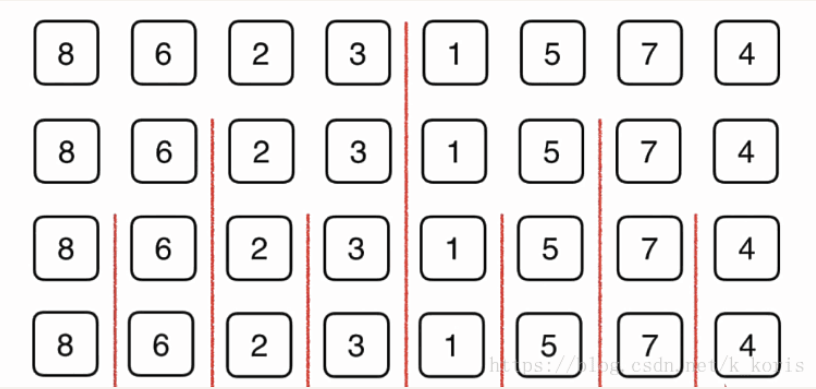
```
public class InsertionSort{
  // Our algorithm class does not allow any instances
   private InsertionSort(){}
 // Use InsertionSort to sort the entire arr array
   public static void sort(Comparable[] arr){
      int n = arr.length;
      for (int i = 0; i < n; i++) {
         Comparable e = arr[i];
         int j = i;
         for( ; j > 0 && arr[j-1].compareTo(e) > 0 ; j--){

            arr[j] = arr[j-1];
         }

         arr[j] = e;
      }
   }
```

1. simplify

2. optimization



Merge Sort

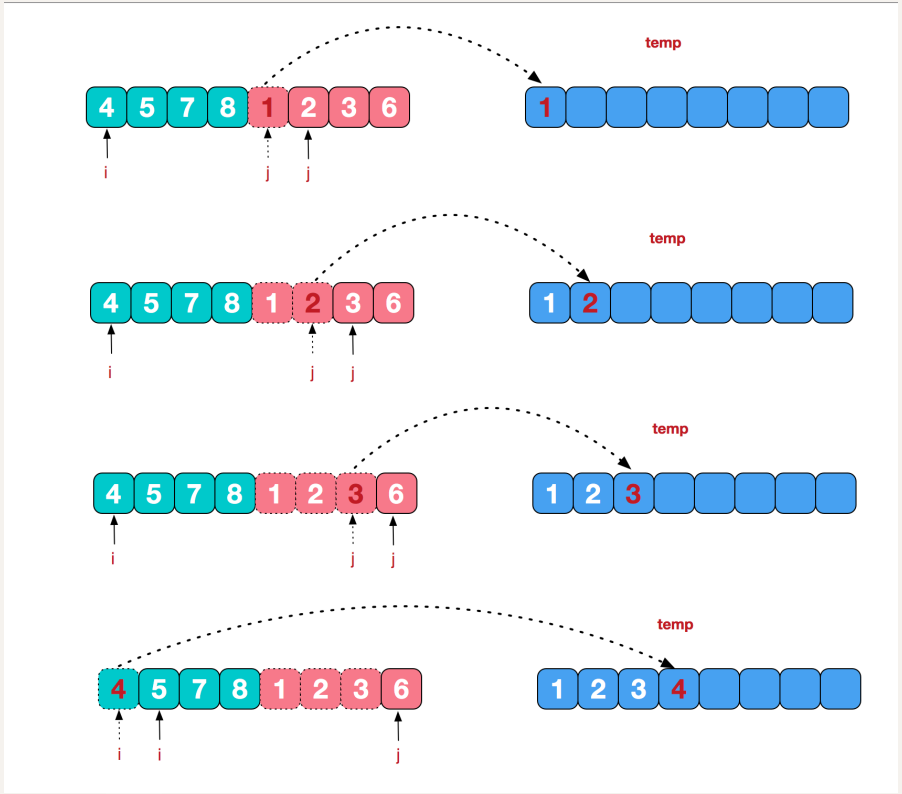


In mergeSort(E[] arr, l, r):

Step 1: Sort arr[l, mid]

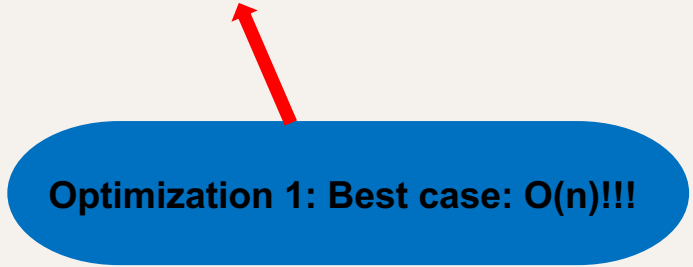
Step 2: Sort arr[mid+1, r]

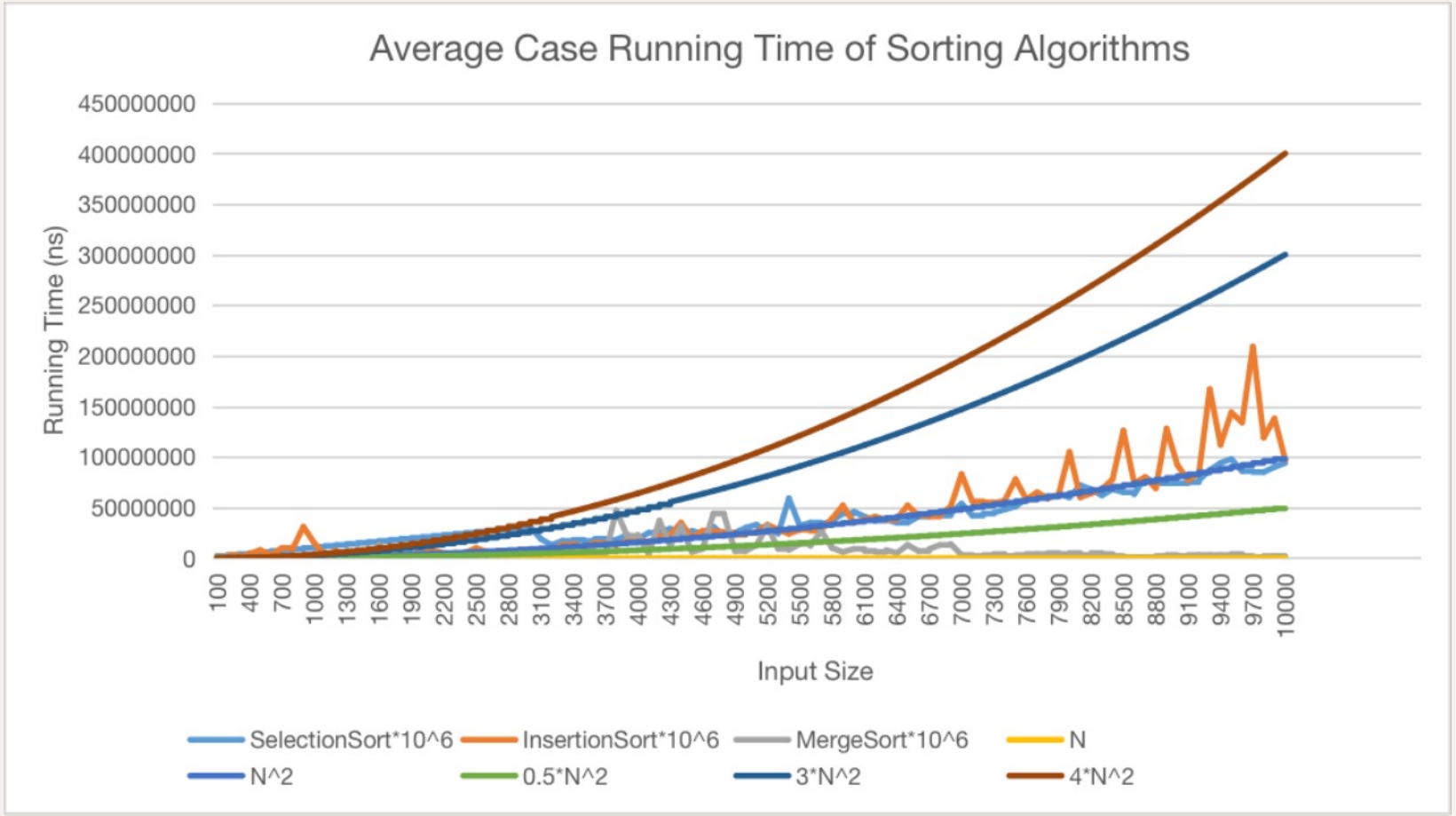
Step 3: Merge sorted arr[l, mid] and arr[mid+1, r]

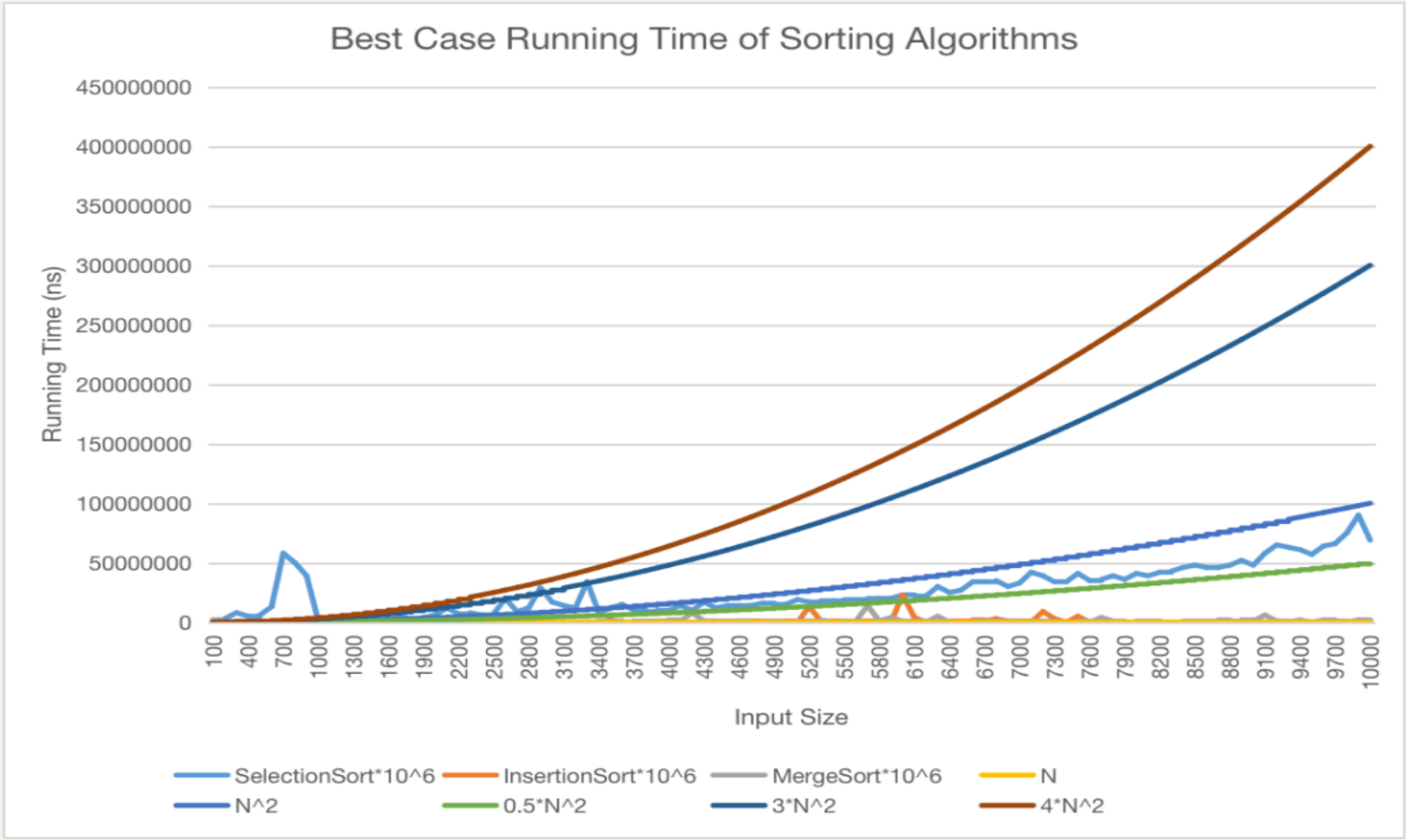

temp
4 5 7 8 1 2 3 6
1
i
j
j
temp
4 5 7 8 1 2 3 6
1 2
i
j
j
temp
4 5 7 8 1 2 3 6
1 2 3
i
j
j
temp
4 5 7 8 1 2 3 6
1 2 3 4
i
i
j

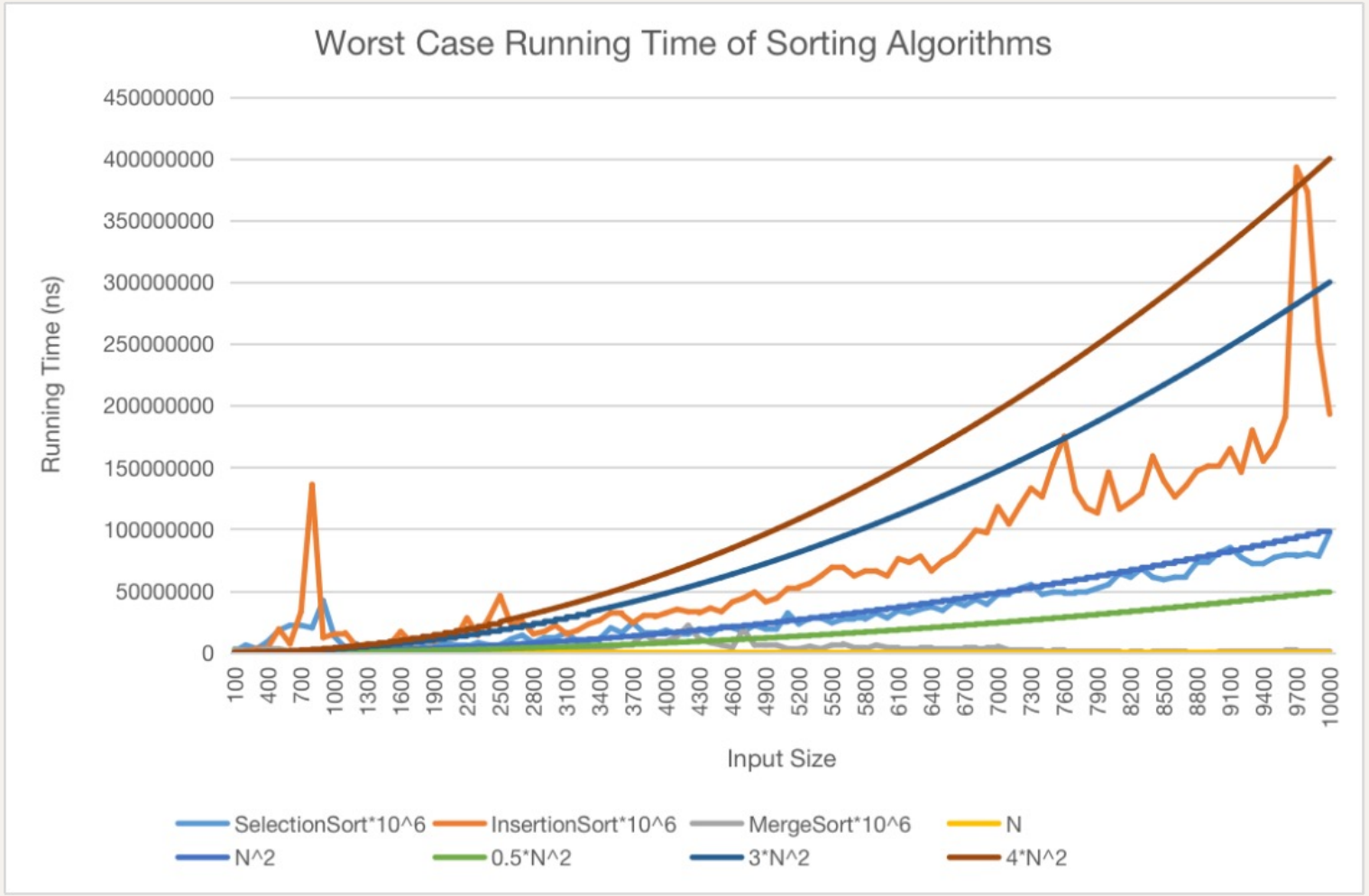
```java
// Combine the two parts of arr[l...mid] and arr[mid+1...r]
   private static void merge(Comparable[] arr, int l, int mid, int r) {
       Comparable[] aux = Arrays.copyOfRange(arr, l, r+1);
    // Initialization, i points to the left half of the starting index position l; j points to the right half of the starting index position mid+1
       int i = l, j = mid+1;
       for( int k = l ; k <= r; k ++ ){
           if( i > mid ){  // If the left half of the elements have all been processed
               arr[k] = aux[j-l]; j ++;
           }
           else if( j > r ){   // If the right half of the elements have all been processed
               arr[k] = aux[i-l]; i ++;
           }
           else if( aux[i-l].compareTo(aux[j-l]) < 0 ){ // The element pointed to by the left half <the element pointed by the right half
               arr[k] = aux[i-l]; i ++;
           }
           else{  // The element referred to in the left half >= the element referred to in the right half
               arr[k] = aux[j-l]; j ++;
           }
       }
   }
```

```java
private static void sort(Comparable[] arr, int l, int r) {
    // Optimization 2: For small-scale arrays, use insertion sort
    if( r - l <= 15 ){
        InsertionSort.sort(arr, l, r);
        return;
    }
    int mid = (l+r)/2;
    sort(arr, l, mid);
    sort(arr, mid + 1, r);

    // Optimization 1: For the case of arr[mid] <= arr[mid+1], do not merge
    // It is very effective for nearly ordered arrays, but for general cases, there is a certain performance loss
    if( arr[mid].compareTo(arr[mid+1]) > 0 )
        merge(arr, l, mid, r);
}
public static void sort(Comparable[] arr){
    int n = arr.length;
    sort(arr, 0, n-1);
}
```

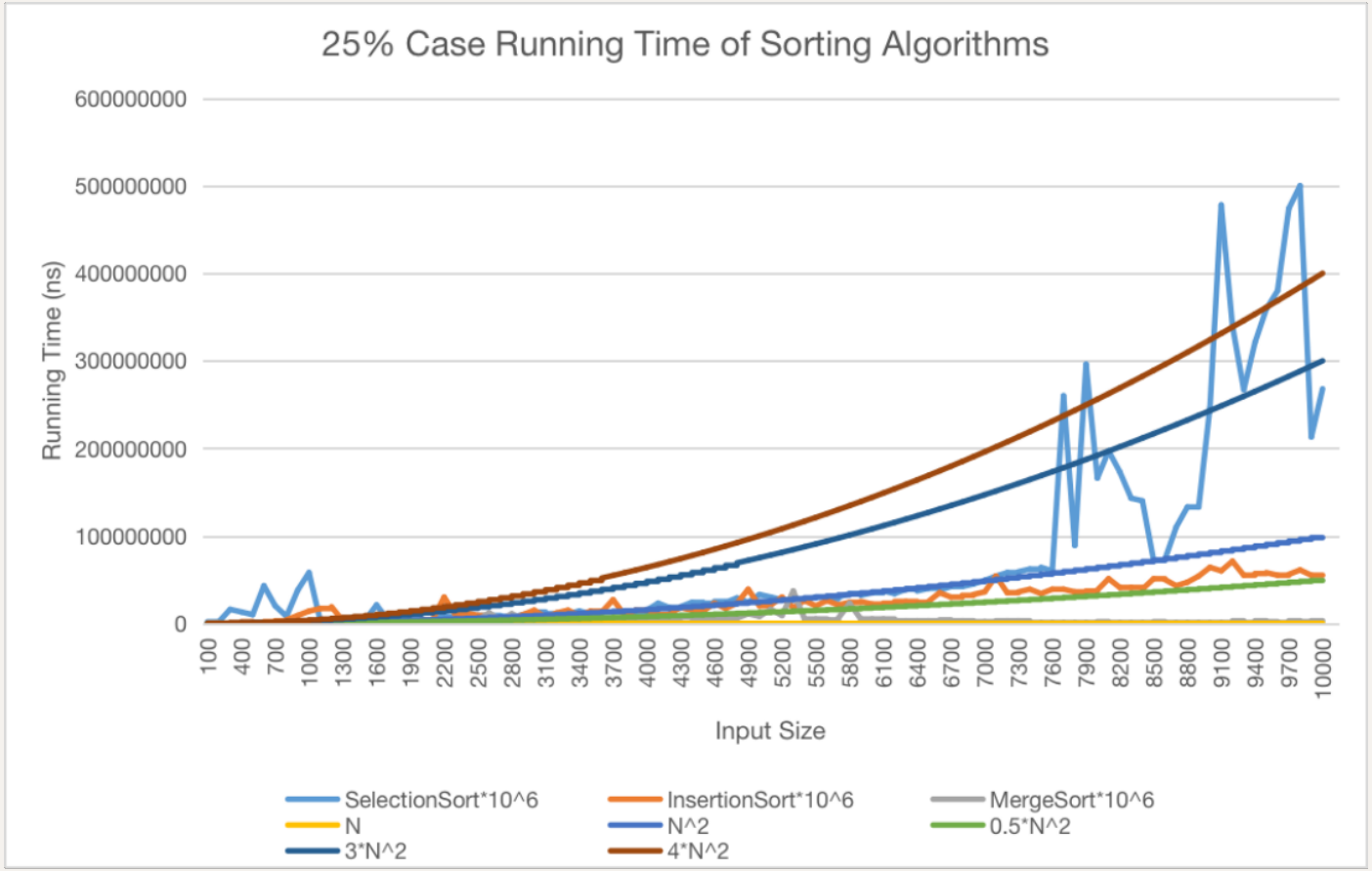


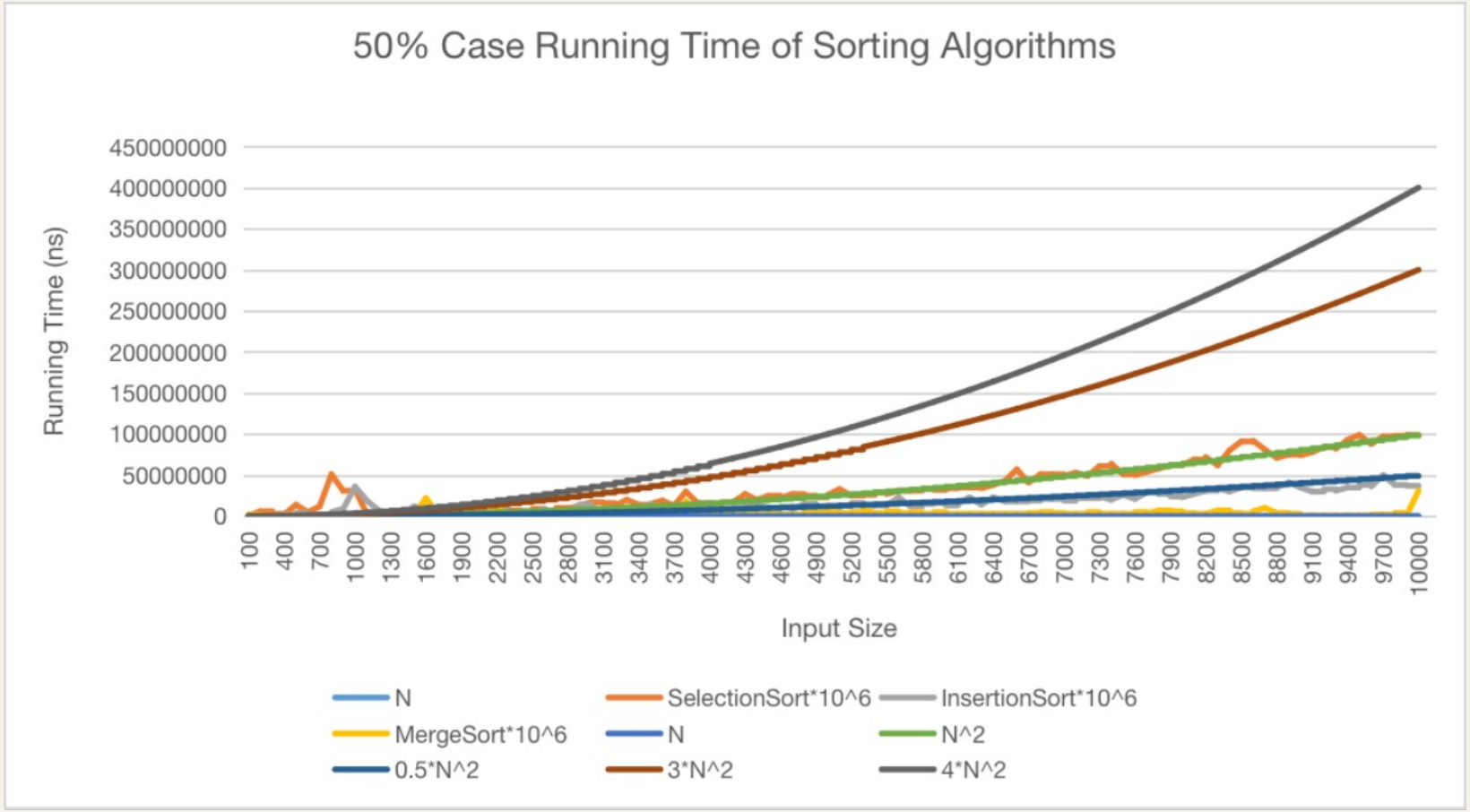

Average Case Running Time of Sorting Algorithms
Running Time (ns)
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
100
400
700
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
Input Size
SelectionSort*10^6
InsertionSort*10^6
MergeSort*10^6
N
N^2
0.5*N^2
3*N^2
4*N^2

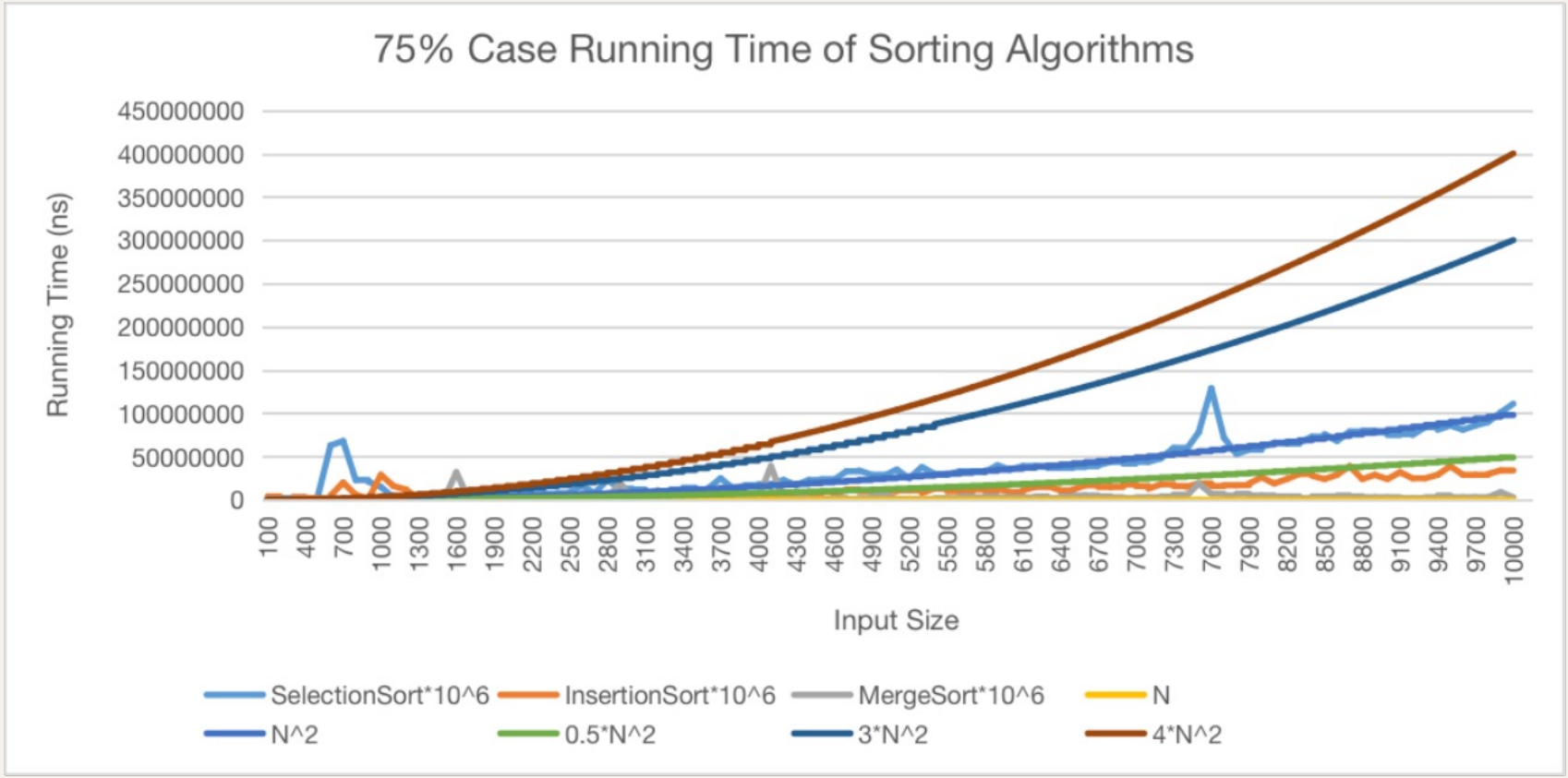

Best Case Running Time of Sorting Algorithms
Running Time (ns)
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
100
400
700
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
Input Size
SelectionSort*10^6
InsertionSort*10^6
MergeSort*10^6
N
N^2
0.5*N^2
3*N^2
4*N^2


Worst Case Running Time of Sorting Algorithms
Running Time (ns)
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
100
400
700
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
Input Size
SelectionSort*10^6
InsertionSort*10^6
MergeSort*10^6
N
N^2
0.5*N^2
3*N^2
4*N^2

```java
// Generate an almost ordered array
// First generate a completely ordered array containing [0...n-1], then randomly exchange swapTimes pairs of data
// swapTimes defines the degree of disorder of the array:
// When swapTimes == 0, the array is completely ordered
// The larger the swapTimes, the more disorder the array tends to be
public static Integer[] generateNearlyOrderedArray(int n, int swapTimes){

    Integer[] arr = new Integer[n];
    for( int i = 0 ; i < n ; i ++ )
        arr[i] = new Integer(i);

    for( int i = 0 ; i < swapTimes ; i ++ ){
        int a = (int)(Math.random() * n);
        int b = (int)(Math.random() * n);
        int t = arr[a];
        arr[a] = arr[b];
        arr[b] = t;
    }

    return arr;
}
```


25% Case Running Time of Sorting Algorithms
600000000
500000000
400000000
300000000
200000000
100000000
0
Running Time (ns)
100
400
700
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
Input Size
SelectionSort*10^6
InsertionSort*10^6
MergeSort*10^6
N
N^2
0.5*N^2
3*N^2
4*N^2


50% Case Running Time of Sorting Algorithms
Running Time (ns)
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
100
400
700
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
Input Size
N
SelectionSort*10^6
InsertionSort*10^6
MergeSort*10^6
N
N^2
0.5*N^2
3*N^2
4*N^2


75% Case Running Time of Sorting Algorithms
Running Time (ns)
450000000
400000000
350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
100
400
700
1000
1300
1600
1900
2200
2500
2800
3100
3400
3700
4000
4300
4600
4900
5200
5500
5800
6100
6400
6700
7000
7300
7600
7900
8200
8500
8800
9100
9400
9700
10000
Input Size
SelectionSort*10^6
InsertionSort*10^6
MergeSort*10^6
N
N^2
0.5*N^2
3*N^2
4*N^2

Questions?

Thanks !