COMP303
Analysis of Algorithms
Project 1: Analysis of Sorting Algorithms

Ertuğrul Yılmaz 041701030

Date: 08 November 2020

Introduction

Sorting algorithms are one of the most interested research topic for computer science. You can find many different algorithms for sorting. All of those algorithms are a product of different logic approaches to same sorting problem. In our time the data is getting bigger and bigger for every service. Except that sorting is a research topic, most importantly it is a problem which have to be solved. All the different approaches about sorting have advantages and disadvantages for different cases. In this project I am going to analyze the most common algorithms for different cases. Algorithms are selection sort, insertion sort, bubble sort, merge sort and randomized hire assistant(not a sorting algorithm). In this report you find the basic idea for each algorithm and the performance evaluation for different types of arrays and different number of elements, also there is explanations for the tools that helps me to reach the goal.

Algorithms

In this section you can the explanations for each algorithms and functions which was used in this project

Creating Array Increasing Order

I used that function to create arrays for the given number of length which has increasing order of elements.

Code:

```
def createArrayIncreasingOrder(n):  #Name of the function and parameter length
    array = []                      #Creation of the empty array.
    for i in range(1, n+1):         #Loop for given number of elements.
        array.append(i)             #Add the to array.
    return array                    #Return the array
```

Figure 1: Increasing Array Algorithm

Creating Array Decreasing Order

I used that function to create arrays for the given number of length which has decreasing order of elements.

Code:

```
def createArrayDecreasingOrder(n):
    array = []
    for i in range(n, 0, -1):
        array.append(i)
    return array
```

#Name of the function and parameter length
#Creation of the empty array.
#Reverse loop for given number of elements.
#Add the to array.
#Return the array

Figure 2: Decreasing Array Algorithm

Creating Array Randomized Order

I used that function to create arrays for the given number of length which has decreasing order of elements.

Code:

```
def createArrayRandomizedOrder(n):
    array = []
    for i in range(n, 0, -1):
        array.append(i)
    for i in range(0, n):
        j=random.randint(i, n-1)
        array[i], array[j] =
array[j], array[i]
    return array
```

#Name of the function and parameter length
#Creation of the empty array.
#Reverse loop
#Filling array with different elements
#Shuffle Loop
#Pick a random index
#Shuffle that index with another index
#Return the array

Figure 3: Randomized Array Algorithm

Holding Counters for Algorithms

For each algorithm I put counters to count the number of rows repeated, number of comparison and number of exchange. I used that counters to make evaluations between different algorithms or the same algorithm. I put counters to the top of every algorithm and named as same, then put counters inside the algorithm. Each algorithm returns the sorted array, number of rows repeated, no of comparisons, number of exchangers.

Code:

```
def insertionSort(array):
    total_Times_row_Repeated = 0
    no_of_comparison = 0
    no_of_exchangers = 0
...
...
...
return array,
total_Times_row_Repeated,
no_of_comparison, no_of_exchangers
```

#Name of the function can be different
#Holds the number of row repeated
#Holds the number of comparison
#Holds the number of exchange
#
#Inside of the function can be different
#
#Returns the sorted array and the counters respectively

Figure 4: Name of counter variables

Selection Sort

The logic between selection sort is simple. It finds the smallest element in the array, then puts exchanges with the first element. After It finds the second smallest and exchanges with the second element of the array then it goes until reach the last element of the array.

Running Time:

```
def selectionSort(array):
    lenght = len(array)
    for j in range(0 , lenght-1):
       smallestIndex = j
        for i in range (j+1, lenght):
            if array[i] <
array[smallestIndex]:
                smallestIndex = i
        array[j], array[smallestIndex]
= array[smallestIndex], array[j]
```

Cost	Time
c1	1
c2	n
c3	n-1
c4	
c5	(n^2)/2 average
c6	
c7	n

Figure 5: Selection Sort Algorithm

$T(n) = c1*1 + c2*n + c3*(n-1) + c4*(n^2)/2 + c5*(n^2)/2 + c6*(n^2)/2 + c7*n = \mathbf{O(n^2)}$

Insertion Sort

Insertion sort starts from the second element of the array then it puts the element which is least right side to correct position. Then gets the third element it goes like that. The left side of the array is stay always sorted if there is an element which need to be sorted ,then it puts the element to the correct position in the left side. Algorithm works until there is no element left on the right side.

Running Time:

$T(n) = c1*n + c2*(n-1) + c3*(n-1) + c4*(n^2)/2 + c5*(n^2)/2 + c6*(n^2)/2 + c7*(n-1) = \mathbf{O(n^2)}$

```
def insertionSort(array):
    for j in range(1, len(array)):
        key = array[j]
        i = j-1
        while i >= 0 and array[i]
> key:
            array[i+1] = array[i]
            i = i-1
        array[i+1] = key
    return array
```

Cost	Time
c1	n
c2	n-1
c3	n-1
c4	$(n^2)/2$ average
c5 →	$(n^2)/2$ average
c6 →	
c7	n-1

Figure 6: Insertion Sort Algorithm

Bubble Sort

Bubble sort is an algorithm which is easy to understand. First Bubble sort holds the beginning index of the array, then it makes sure that index has the right element. To achieve that is sort from the last element of the array it exchanges with the elements which is in the right side of the index. When index has the right element it increments the current index to next and goes with the same process until current index becomes the last index of the array.

Running Time:

```
def bubbleSort(array):
    for i in range(0, len(array)):
       for j in range(len(array)-1, i, -1):
            if array[j] < array[j-1]:
                array[j], array[j-1] = array[j-1], array[j]
       return array
```

Cost	Time
c1	n
c2	n-1
c3	$(n^2)/2$ average
c4	$(n^2)/2$ average

Figure 7: Bubble Sort Algorithm

$T(n) = c1*n + c2*(n-1) + c3*(n^2)/2 + c4*(n^2)/2 = \mathbf{O(n^2)}$

Merge Sort

Merge is way more different than other algorithms and it also has more complexity inside. It takes an array and divides into subproblems, then it also divides that subproblems into smaller subproblems until subproblems has the length of one or two. Array becomes like a binary tree. It compares the leaf between them, then combines them in the correct order until there is no leaf which was not combined. In brief it divides the array, then combines it in the correct order.

Running Time:

```
def merge(array, p, q, r):                  Cost        Time
    n1 = q-p+1                              c1          1
    n2 = r-q                                c2          1
    L = []                                  c3          1
    R = []                                  c4          1
    for i in range(0, n1):                  c5          ≈ n/2
        L.append(array[p + i])              c6          ≈ n/2
    L.append(math.inf)                      c7          1
    for j in range(0, n2):                  c8          ≈ n/2
        R.append(array[q + 1 + j])          c9          ≈ n/2
    R.append(math.inf)                      c10         1
    i = 0                                   c11         1
    j = 0                                   c12         1
    for k in range(p, r+1):                 c13         n
        if L[i] <= R[j]:                    c14         max n times
            array[k] = L[i]                 c15         max n times
            i += 1                          c16         max n times
        else:                               c17         max n times
            array[k] = R[j]                 c18         max n times
            j += 1                          c19         max n times
```

Figure 8: Merge Operation Algorithm

$T_1(n) = (c1+c2+c3+c4+c7+c10+c11+c12)*1 + (c13+c14+c15+c16+c17+c18+c19)*n + (c5+c6+c8+c9)*(n/2) =$ **O(n)**

```
def mergeSort(array, p, r):                 Cost        Time
    if p < r:                               c1          1
        q = (p + r) // 2                    c2          1
        mergeSort(array, p, q)              c3          O(lgn) * O(n)
        mergeSort(array, q + 1, r)          c4          O(lgn) * O(n)
        merge(array, p, q, r)               c5          O(n)
    return array
```

Figure 9: Merge Sort Algorithm

$T_2(n) = (c1+c2)*1 + 2*(O(nlgn)) + O(n) =$ **O(nlgn)**

We divide the array lg(n) times and merge function has O(n) = O(n*lgn)

Randomized Hire Assistant

Randomized Hire Assistant is not sorting algorithm. It takes the first element as best then looks the second element if it is better than best then it changes best with it. It looks every element of the array from first to last.

Running Time:

```
def hireAssistant(array):
    best = 0
    for i in range(0, len(array)):
        if array[i] > best:
            best = array[i]
    return array
```

Cost	Time
c1	1
c2	n
c3	n-1
c4	n-1

Figure 10: Hire Assistant Algorithm

T(n) = c1*1+ c2*n + c3*(n-1) + c4*(n-1) = **O(n)**

Comparisons

I have done my comparisons according to two main topic. First one is number of elements, I analyzed the algorithms for n=10,100,1000 and 10000, I couldn't done it for 100000 elements because It was taking minimum 30 minutes to sort the array. Also I didn't print the chart as bar charts not plot because of number of elements I may seem not good to me. It is way better to analyze with bar charts.

Algorithm for Different Number of Elements

Code:

```
def compareIncreasing1():
total_Times_row_Repeated_Hire = [0] * 5 #Index[0] for n=10, index[1] for
n=100…
no_of_comparison_Hire = [0] * 5#Index[0] for n=10, index[1] for n=100…
no_of_exchangers_Hire = [0] * 5#Index[0] for n=10, index[1] for n=100…
#Same for other algorithms
total_Times_row_Repeated_Insertion = [0] * 5
no_of_comparison_Insertion = [0] * 5
no_of_exchangers_Insertion = [0] * 5

total_Times_row_Repeated_Selection = [0] * 5
no_of_comparison_Selection = [0] * 5
```

```
no_of_exchangers_Selection = [0] * 5
total_Times_row_Repeated_Bubble = [0] * 5
no_of_comparison_Bubble = [0] * 5
no_of_exchangers_Bubble = [0] * 5

total_Times_row_Repeated_Merge = [0] * 5
no_of_comparison_Merge = [0] * 5
no_of_exchangers_Merge = [0] * 5

# 10, 100, 1000, 10000, 50000
#Variable n and index are related to each other
n = 10
index = 0
#Copying array to give the same array to all algorithms
arrayHire = createArrayIncreasingOrder(n)
arrayInsertion = arrayHire.copy()
arraySelection = arrayHire.copy()
arrayBubble = arrayHire.copy()
arrayMerge = arrayHire.copy()
#Array which is going be sorted
print("\nN=", n)
print("Array:", arrayHire)
#Storing the return values of the arrays
sortedArrayHire, total_Times_row_Repeated_Hire[index],
no_of_comparison_Hire[index], no_of_exchangers_Hire[
    index] = hireAssistant(arrayHire)
#Storing the return values of the arrays
sortedArrayInsertion, total_Times_row_Repeated_Insertion[index],
no_of_comparison_Insertion[index], \
no_of_exchangers_Insertion[index] = insertionSort(arrayInsertion)
#Storing the return values of the arrays
sortedArraySelection, total_Times_row_Repeated_Selection[index],
no_of_comparison_Selection[index], \
no_of_exchangers_Selection[index] = selectionSort(arraySelection)
#Storing the return values of the arrays
sortedArrayBubble, total_Times_row_Repeated_Bubble[index],
no_of_comparison_Bubble[index], no_of_exchangers_Bubble[
    index] = bubbleSort(arrayBubble)
```

```
#Storing the return values of the arrays
sortedArrayMerge, total_Times_row_Repeated_Merge[index],
no_of_comparison_Merge[index], no_of_exchangers_Merge[
    index] = mergeSort(arrayMerge, 0, len(arrayMerge)-1)
#Printing the sorted arrays
print("Hire:", sortedArrayHire, " ", total_Times_row_Repeated_Hire[index], "
", no_of_comparison_Hire[index], " ",
      no_of_exchangers_Hire[index], " ")
print("Insertion:", sortedArrayInsertion, " ",
total_Times_row_Repeated_Insertion[index], " ",
      no_of_comparison_Insertion[index], " ",
no_of_exchangers_Insertion[index], " ")
print("Selection:", sortedArraySelection, " ",
total_Times_row_Repeated_Selection[index], " ",
      no_of_comparison_Selection[index], " ",
no_of_exchangers_Selection[index], " ")
print("Bubble:", sortedArrayBubble, " ",
total_Times_row_Repeated_Bubble[index], " ",
      no_of_comparison_Bubble[index], " ", no_of_exchangers_Bubble[index], "
")
print("Merge:", sortedArrayMerge, " ", total_Times_row_Repeated_Merge[index],
" ",
      no_of_comparison_Merge[index], " ", no_of_exchangers_Merge[index], " ")
#Clearing the arrays for a new sort operation
arrayHire.clear()
arrayInsertion.clear()
arraySelection.clear()
arrayBubble.clear()
arrayMerge.clear()
sortedArrayHire.clear()
sortedArrayInsertion.clear()
sortedArraySelection.clear()
sortedArrayBubble.clear()
sortedArrayMerge.clear()
#Same process for n=100
# 100
n = 100
index = 1
.........
```

```
..........

..........

..........

# Printing of the results as bar chart

names = ['Hire', 'Insertion', 'Selection', 'Bubble', 'Merge']


plt.figure(figsize=(15, 7))

# n=10

index = 0


plt.subplot(4, 3, 1)

plt.title('Total Number of\nRows Repeated')

plt.ylabel('N=10')

plt.bar(names, [total_Times_row_Repeated_Hire[index],
total_Times_row_Repeated_Insertion[index],

                total_Times_row_Repeated_Selection[index],
total_Times_row_Repeated_Bubble[index],
total_Times_row_Repeated_Merge[index]],

        color=['b', 'r', 'm', 'g', 'y'])

plt.subplot(4, 3, 2)

plt.title('Number of Comparisons')

plt.bar(names, [no_of_comparison_Hire[index],
no_of_comparison_Insertion[index],

                no_of_comparison_Selection[index],
no_of_comparison_Bubble[index], no_of_comparison_Merge[index]],

        color=['b', 'r', 'm', 'g', 'y'])

plt.subplot(4, 3, 3)

plt.title('Number of Exchanges')

plt.bar(names, [no_of_exchangers_Hire[index],
no_of_exchangers_Insertion[index],

                no_of_exchangers_Selection[index],
no_of_exchangers_Bubble[index],no_of_exchangers_Merge[index]],

color=['b', 'r', 'm', 'g', 'y'])

# n=100

.......... # Same for the rest of the number of elements

...........

plt.suptitle('Array Type: Increasing', fontweight='bold')

plt.show()
```

Figure 11: Algorithm of compareIncreasing

I did the same process for different array types too.

Algorithm for Different Types of Arrays

Code:

```
def compareArrayTypes(n):
    arrayIncreased = createArrayIncreasingOrder(n)     #Increasing Array
    arrayDecreased = createArrayDecreasingOrder(n) #Decreasing Array
    arrayRandomized = createArrayRandomizedOrder(n) #Decreasing Array

    total_Times_row_Repeated_Hire = [0] * 5 #Index[0] for n=increasing,
index[1] for n=decreasing, index[2] for n=randomized
    no_of_comparison_Hire = [0] * 5 #Index[0] for n=10, index[1] for n=100,
index[2] for n=randomized
    no_of_exchangers_Hire = [0] * 5 #Index[0] for n=10, index[1] for n=100,
index[2] for n=randomized
#Same for other algorithms
    total_Times_row_Repeated_Insertion = [0] * 5
    no_of_comparison_Insertion = [0] * 5
    no_of_exchangers_Insertion = [0] * 5

    total_Times_row_Repeated_Selection = [0] * 5
    no_of_comparison_Selection = [0] * 5
    no_of_exchangers_Selection = [0] * 5

    total_Times_row_Repeated_Bubble = [0] * 5
    no_of_comparison_Bubble = [0] * 5
    no_of_exchangers_Bubble = [0] * 5

    total_Times_row_Repeated_Merge = [0] * 5
    no_of_comparison_Merge = [0] * 5
    no_of_exchangers_Merge = [0] * 5

    #Getting the test results and algorithm returns for Increasing Array
    index = 0

    tempArray = arrayIncreased.copy()
    sortedArrayHire, total_Times_row_Repeated_Hire[index],
no_of_comparison_Hire[index], no_of_exchangers_Hire[index] =
hireAssistant(tempArray)
    tempArray.clear()
```

```
    tempArray = arrayIncreased.copy()
    sortedArrayInsertion, total_Times_row_Repeated_Insertion[index],
no_of_comparison_Insertion[index], no_of_exchangers_Insertion[index] =
insertionSort(tempArray)
    tempArray.clear()

    tempArray = arrayIncreased.copy()
    sortedArraySelection, total_Times_row_Repeated_Selection[index],
no_of_comparison_Selection[index], no_of_exchangers_Selection[index] =
selectionSort(tempArray)
    tempArray.clear()

    tempArray = arrayIncreased.copy()
    sortedArrayBubble, total_Times_row_Repeated_Bubble[index],
no_of_comparison_Bubble[index], no_of_exchangers_Bubble[index]
=bubbleSort(tempArray)
    tempArray.clear()

    tempArray = arrayIncreased.copy()
    sortedArrayMerge, total_Times_row_Repeated_Merge[index],
no_of_comparison_Merge[index], no_of_exchangers_Merge[index] =
mergeSort(tempArray, 0, len(tempArray)-1)
    tempArray.clear()

    sortedArrayHire.clear()
    sortedArrayInsertion.clear()
    sortedArraySelection.clear()
    sortedArrayBubble.clear()
    sortedArrayMerge.clear()

    # Same Process for Decreasing Array
    index = 1
••••••••••••
••••••••••••
    # Same Process for Randomized Array
    index = 2
••••••••••••
••••••••••••
```

```
    names = ['Hire', 'Insertion', 'Selection', 'Bubble', 'Merge']
    plt.figure(figsize=(15, 6))
    # Plottinn for Increasing Array
    index = 0
    plt.subplot(3, 3, 1)
    plt.title('Total Number of\nRows Repeated')
    plt.ylabel('Increasing\nArray', fontweight='bold')
    plt.bar(names, [total_Times_row_Repeated_Hire[index],
total_Times_row_Repeated_Insertion[index],
                    total_Times_row_Repeated_Selection[index],
total_Times_row_Repeated_Bubble[index],
                    total_Times_row_Repeated_Merge[index]],
            color=['b', 'r', 'm', 'g', 'y'])
    plt.subplot(3, 3, 2)
    plt.title('Number of Comparisons')
    plt.bar(names, [no_of_comparison_Hire[index],
no_of_comparison_Insertion[index],
                    no_of_comparison_Selection[index],
no_of_comparison_Bubble[index], no_of_comparison_Merge[index]],
            color=['b', 'r', 'm', 'g', 'y'])
    plt.subplot(3, 3, 3)
    plt.title('Number of Exchanges')
    plt.bar(names, [no_of_exchangers_Hire[index],
no_of_exchangers_Insertion[index],
                    no_of_exchangers_Selection[index],
no_of_exchangers_Bubble[index], no_of_exchangers_Merge[index]],
            color=['b', 'r', 'm', 'g', 'y'])
```

```
    # Same plotting for Decreasing Array
    index = 1
    ...........
    # Same plotting for Randomized Array
    index = 2
    ...........
    title = 'Number of elements: ' + str(n)
    plt.suptitle(title, fontweight='bold')
    plt.show()
```

Figure 12: Algorithm of compareArrayTypes

I toke the element number as function parameter.

Analyze for Different Number of Elements

When you analyzing the algorithms for different number of elements, you have three different array type to choose. First one is increasing array type, second one is decreasing array type and third one is randomized array. Before analyzing the result you can see charts below.

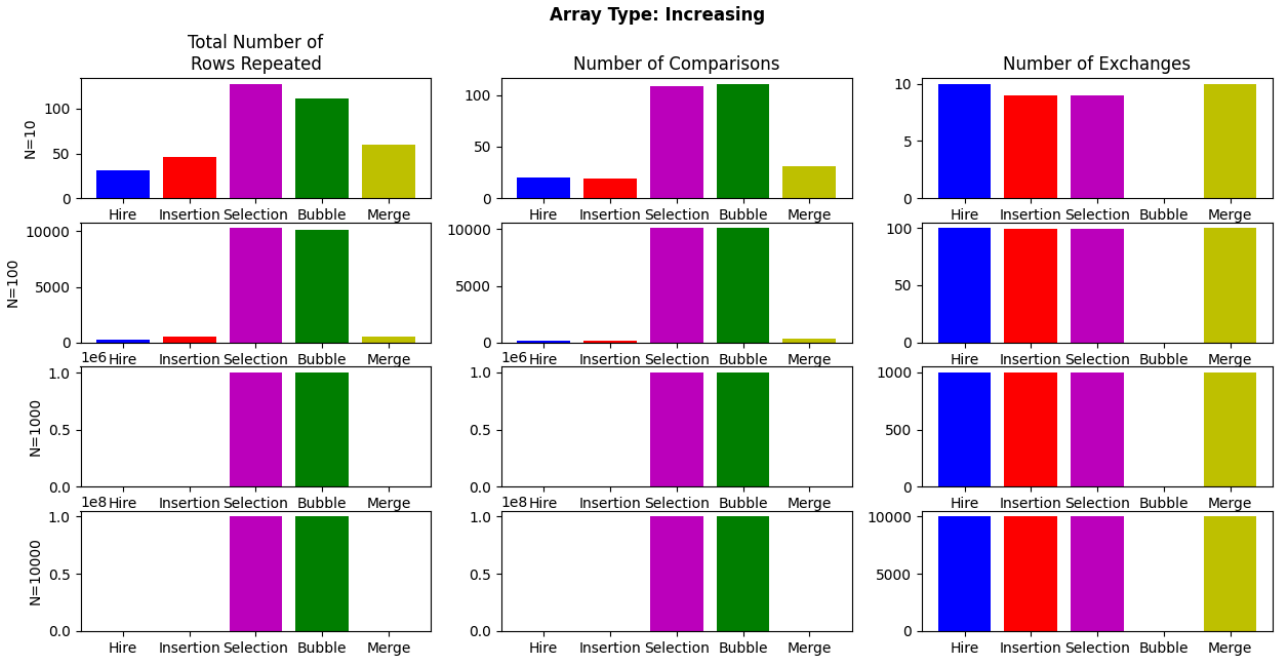

Figure 13: Chart of Increasing Array Compare

When we choose the array type increasing for number of elements. We can see the most two efficient sorting algorithms are insertion and merge. When we look at the number of comparisons, insertion and merge has the best values and bubble has the worts value. At number of exchanges bubble makes 0 exchange that means bubble owes the comparisons that it made. Also we can say that selection is bad choice. Number of exchanges of hiring algorithm is also high because it was an increasing array and every next candidate has the better value than the old. Number of exchanges of insertion is high because it always store key value in the array.

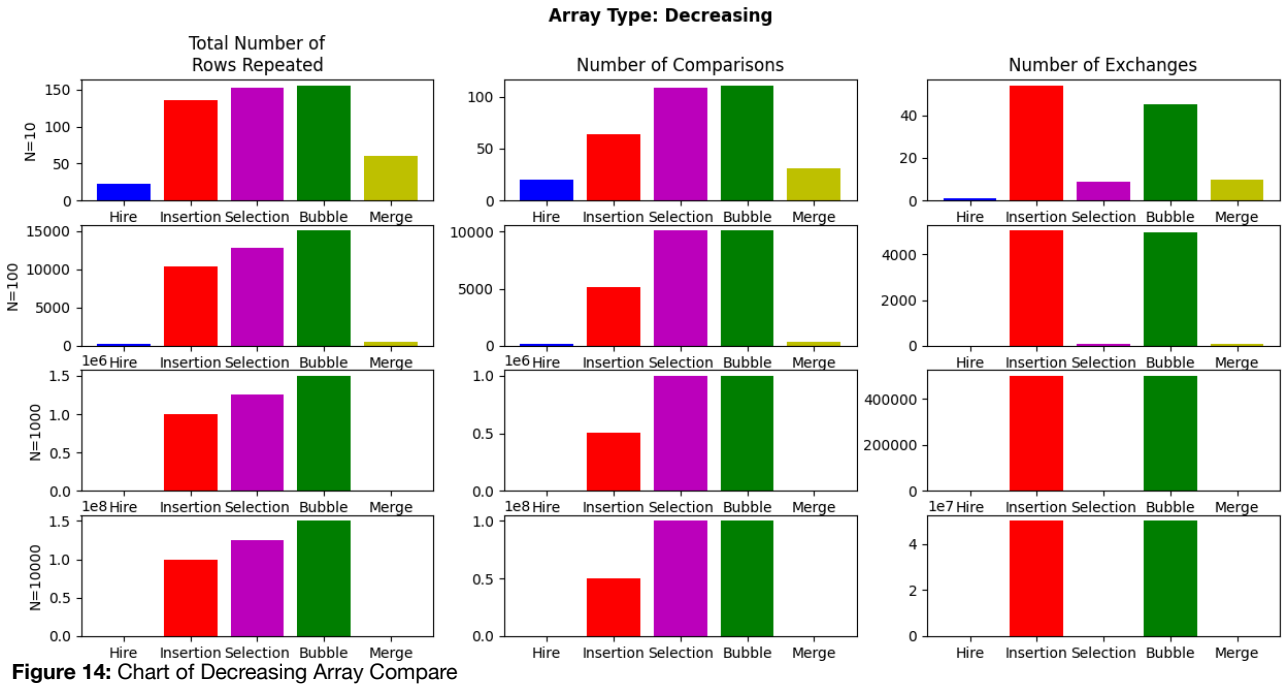

Figure 14: Chart of Decreasing Array Compare

When we changed array type to decreasing for number of element, hiring algorithm has the best value on all categories because the first candidate is best one. Merge has the best values among all of the sorting algorithms except selection with exchanges. It is much more easy to turn array upside down for merge because it divides the array into small pieces and combines them into small to big. The number of comparison of bubble is increased because every that it compares is smaller than the previous. Selection has high comparison but has the smallest exchange because it is hard to find the next small element in but easy exchange it to right place. Number of comparison for insertion is proportionally increased because the more it exchanges there will be less comparison. Bubble has the worst total number of row repeated because it always has to travel the rest of the array and make changes every time.

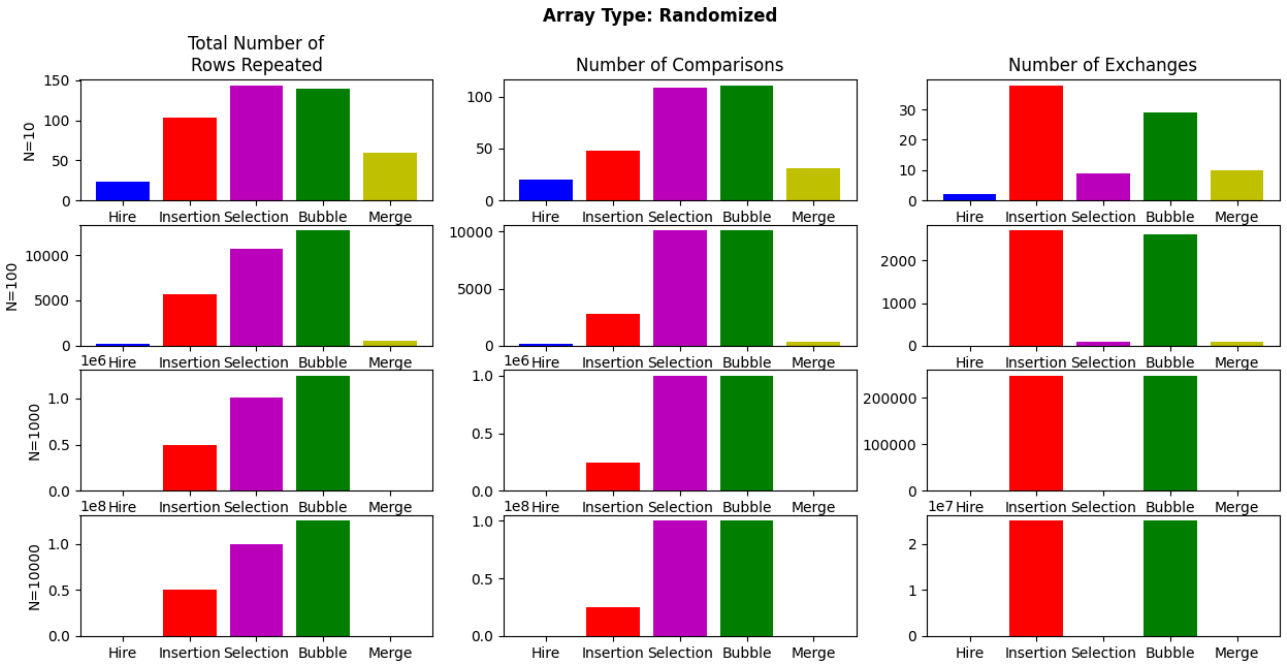

Figure 15: Chart of Randomized Array Compare

When we changed array to randomized, I think there is no need to discuss about hiring algorithm because it is not sorting. Merge has the best values among all of the sorting algorithms with no exception. Insertion has the second best total number of rows repeated and number of comparisons but it has the worst number of exchange because it always exchanges with key does not matter even if the element is in the right place or not. Selection sort is good with exchange because it does need to look for place which element going to be putted, It just looks for the smallest element and puts to the current place. We can say that bubble is not good for any category. The merge is the best for randomized arrays.

Analyze for Different Types of Arrays

Another way to analyze the algorithms is changing the array types. In my code I took the number of elements as a parameter and for given number I printed the algorithms for different array types. The array types that I analyzed are increasing which has increasing order of elements, decreasing which has decreasing order of elements and randomized which has the random order of element. The chart that I achieve is below.

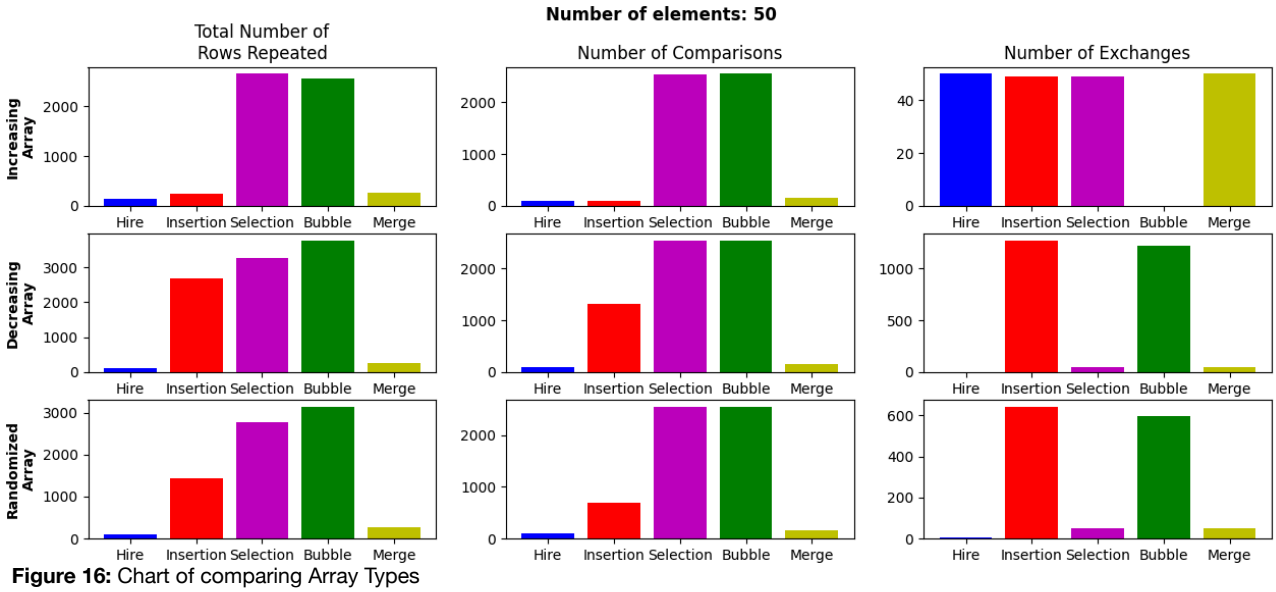

Figure 16: Chart of comparing Array Types

When we look at the we can see that Merge always has the best total number of rows repeated than others. Merge sort is just bad for increasing arrays because it always divides and combines, it doesn't matter array is ordered or not. Insertion sort always has high exchange value but it is better when comparing a sorted array. Bubble sort is good for increasing arrays when exchanging. Selection can be useful if exchange matters while working with decreasing or increasing arrays.

GUI

Also my program has a simple user interface with a person does not need know anything about the function or variables to run code. Any person can see the evaluations by just pressing run and using the interface. Example screenshots are below:

```
Welcome to Array and Algorithm Comparing Program

What dou you want to compare?(Type 1 or 2 or 3)
1. Different Number of Array Types
2. Different Number of Elements
3. Exit
1
How many elements you want to compare?
50
```

Figure 17: GUI Example

Gives

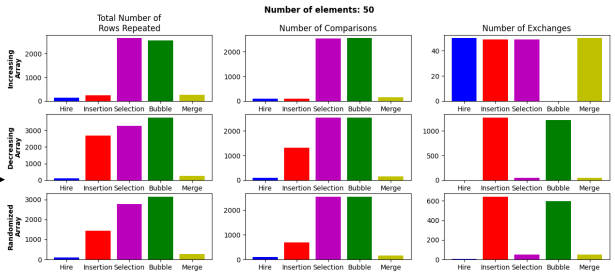

Figure 18: Example Chart

```
What dou you want to compare?(Type 1 or 2 or 3)
1. Different Number of Array Types
2. Different Number of Elements
3. Exit
2
Which Array type?(Type 1 or 2 or 3)
1. Increasing Array
2. Decreasing Array
3. Randomized Array
1
```

Figure 19: GUI Example

Gives

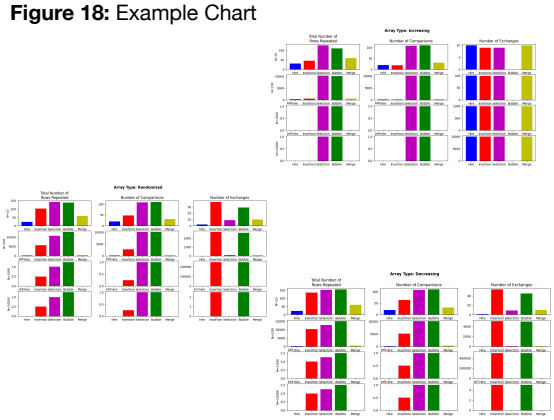

Figure 20: Example Charts

```
What dou you want to compare?(Type 1 or 2 or 3)
1. Different Number of Array Types
2. Different Number of Elements
3. Exit
a
IT IS NOT A VALID INPUT!
```

Figure 21: GUI Error Handling

```
What dou you want to compare?(Type 1 or 2 or 3)
1. Different Number of Array Types
2. Different Number of Elements
3. Exit
3

Process finished with exit code 0
```

Figure 22: Exit from GUI

References

1. COMP303 Analysis of Algorithm Lecture Notes
2. https://matplotlib.org/tutorials/introductory/pyplot.html

All Code

https://colab.research.google.com/drive/19qM_OVPw5gq-HQa7Owp7TS3b98WT0Udr?usp=sharing

https://drive.google.com/file/d/1BRaRt_jOxmrn8IBK2jXEZWqS5mQNn2aR/view?usp=sharing