

# 

# 

# CZ2001 Algorithm

Example Class 3 Report

|  |  |
| --- | --- |
| Members - SE1 (GROUP 2) | |
| U1620986A | Lee Yi Zhuo |
| U1721978B | Ngo Jun Hao Jason |
| U1722308D | Thomas Stephen Felix |
| U1722841L | Tan Yong Heng Kenrick |
| N1804770J | Yung-Hsueh Lee |

Implementation of Insertion Sort

Description:

Insertion sort is a sorting algorithm in which the elements are transferred one at a time to the right position.

Code used for Insertion Sort:

**void InsertionSort (ALIST slot[ ], int n)**

{

for (int i=1; i < n; i++) {

for (int j=i; j > 0; j--) {

if (slot[j].key < slot[j-1].key)

swap(slot[j], slot[j-1]);

else break;

}

}

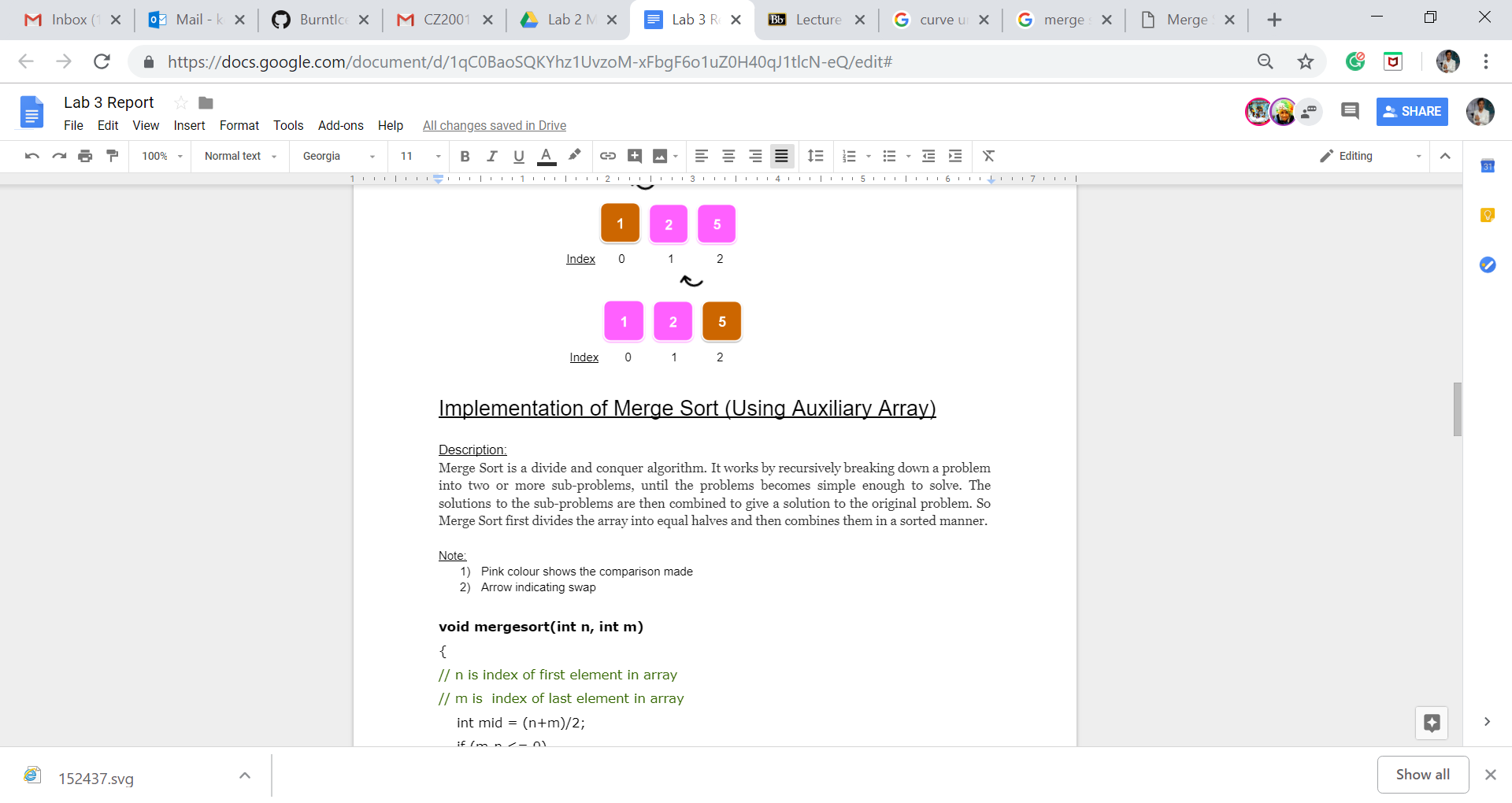
}

Visualisation of /insertion Swap:(Only for presentation Slide)

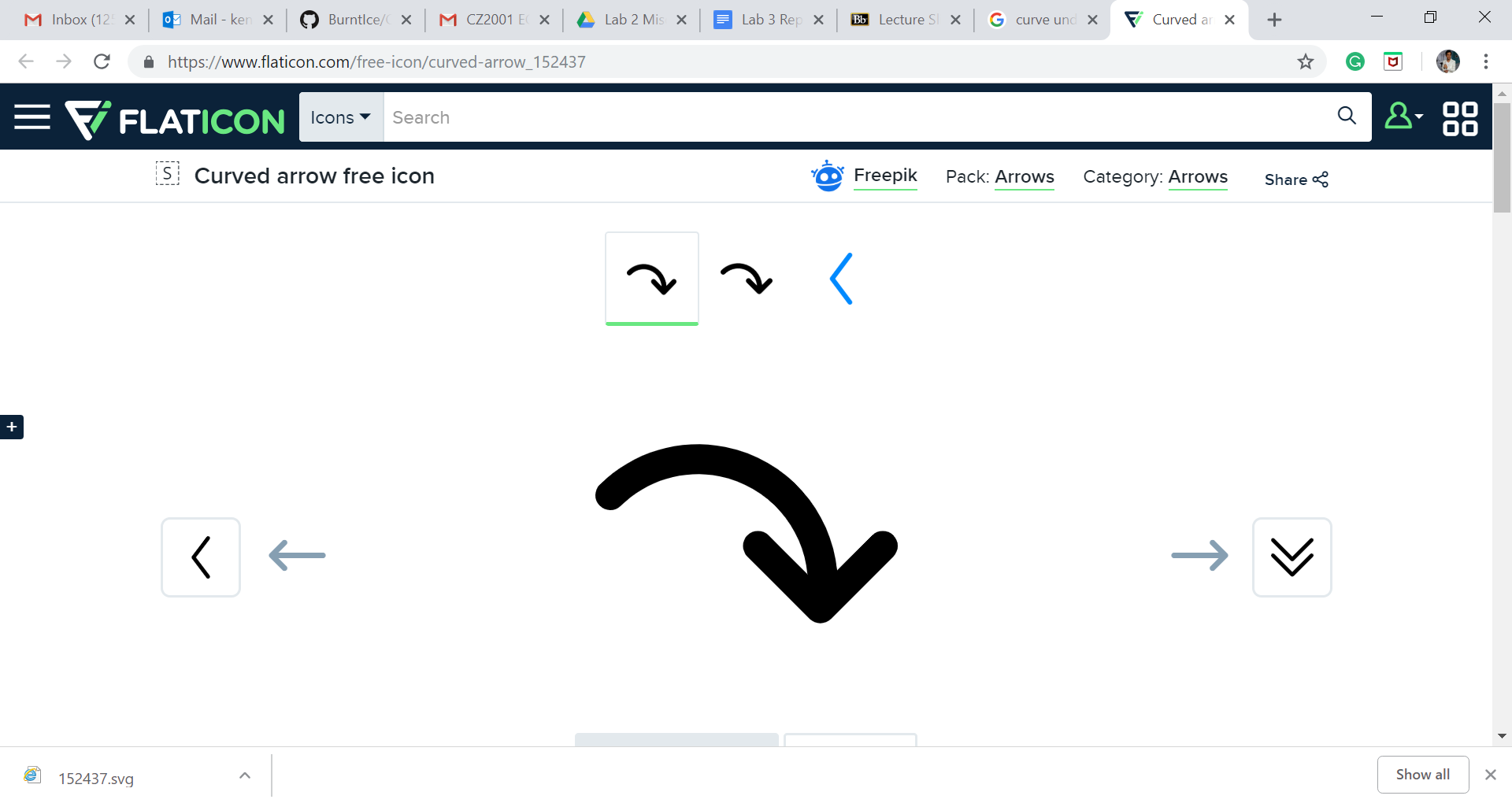
ALIST:



Index 0 1 2

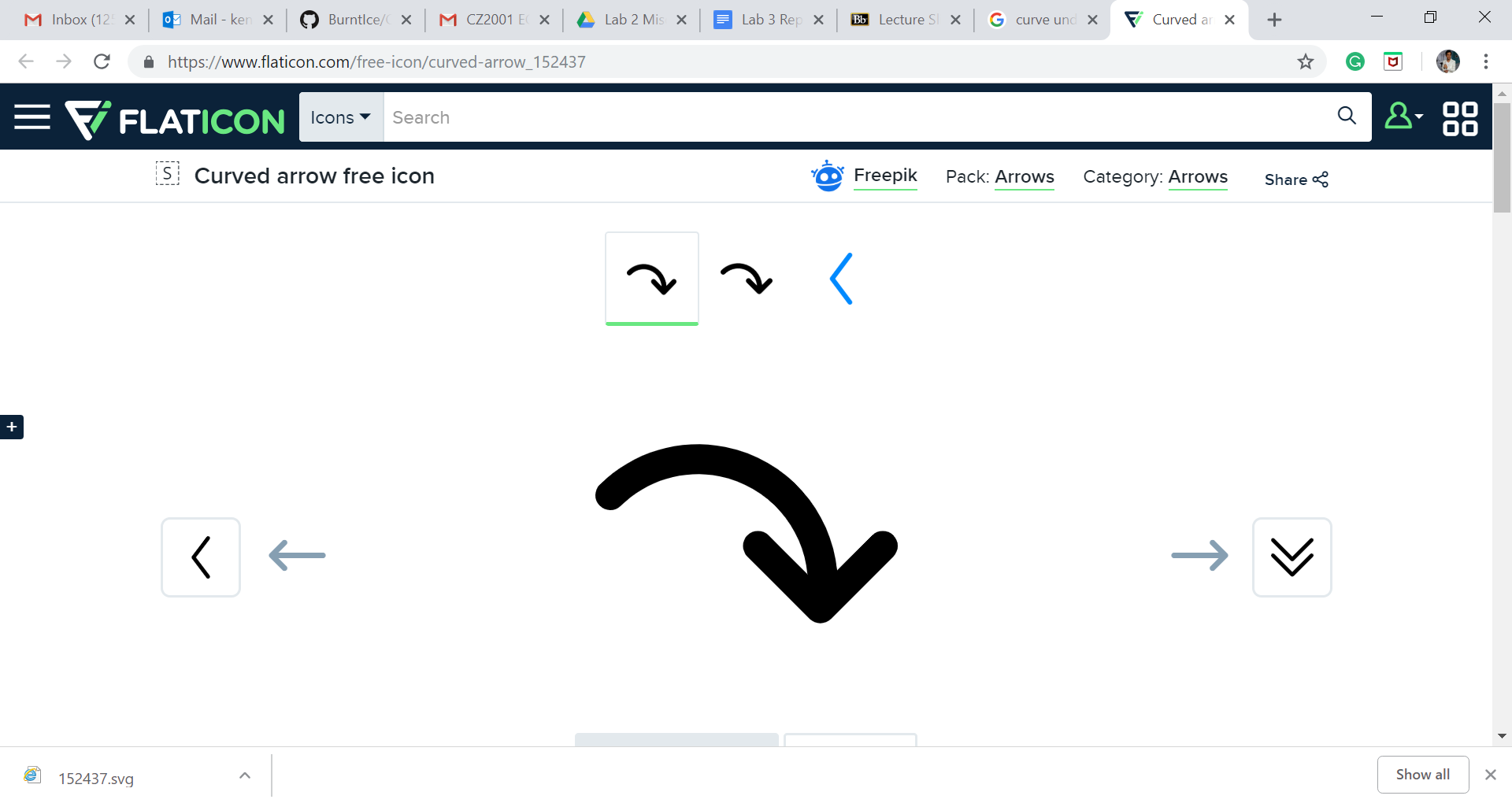
 

Index 0 1 2





Index 0 1 2





Index 0 1 2

Implementation of Merge Sort

Description:

Merge Sort is a divide and conquer algorithm. It works by recursively breaking down a problem into two or more sub-problems, until the problems becomes simple enough to solve. The solutions to the sub-problems are then combined to give a solution to the original problem. So mergesort(int n, int m) first divides the array into equal halves and then combines them in merge(int n, int m) a sorted manner.

Code used for Merge Sort:

**void mergesort(int n, int m)**

{ // n is index of first element in array

// m is index of last element in array

int mid = (n+m)/2;

if (m-n <= 0)

return;

else if (m-n > 1) {

mergesort(n, mid);

mergesort(mid+1, m);

}

merge(n, m);

}

**void merge(int n, int m)**

{

int mid = (n+m)/2;

int a = n, b = mid+1, i, tmp;

if (m-n <= 0)

return;

while (a <= mid && b <= m) {

cmp = compare(slot[a], slot[b]);

if (cmp > 0) { //slot[a] > slot[b]

tmp = slot[b++];

for (i = ++mid; i > a; i--)

slot[i] = slot[i-1];

slot[a++] = tmp;

}

else if (cmp < 0) //slot[a] < slot[b]

a++;

else { //slot[a] == slot[b]

if (a == mid && b == m)

break;

tmp = slot[b++];

a++;

for (i = ++mid; i > a; i--)

slot[i] = slot[i-1];

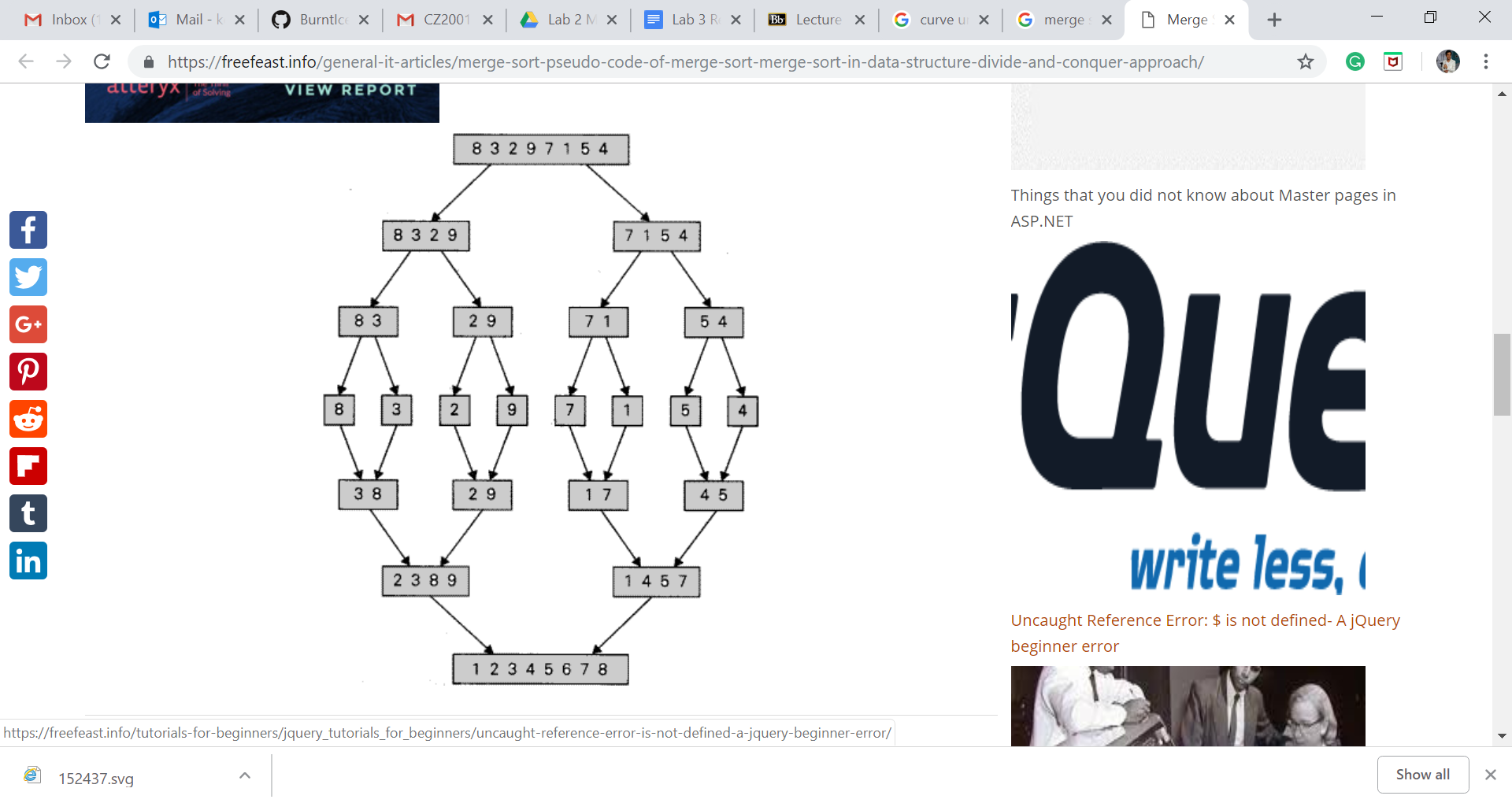
slot[a++] = tmp;

}

} // end of while loop;

} // end of merge

Visualisation of MergeSort: (Only for presentation)



Comparison Table

1. Randomly generated datasets
2. Integers in ascending order
3. Integers in descending order

Theoretical Analysis of the Time Complexity of Insertion Sort

1. **Best Case [ *n*-1 = O(*n*) ]**

The best case for insertion sort occurs when the array is already sorted in the desired order. In this case, each key in the array, except for the first key, only needs to be compared to the key right before it. So for an array of length *n*, only *n*-1 number of key comparisons need to be made, resulting in a time complexity of O(*n*).

1. **Worst Case [ *n*\*(*n*-1) /2 = O(*n*2) ]**

The worst case for insertion sort occurs when the array is already sorted, but in the reverse order. In this case, each key in the array, except for the first key, needs to be compared to all the keys that come before it.

For a key at index *i*, there are *i*-1 keys before it, so it has to go through *i*-1 number of key comparisons. If there are *n* number of keys in the array (i.e. array of size n), there will be (*n*-1) + (*n*-2) + … + 2 + 1 = (*n*\*(*n*-1) /2) number of key comparisons when sorting through this array, resulting in a time complexity of O(*n*^2).

1. **Average Case [ *n*\*(*n*-1) / 4 = O(*n*2) ]**

The average case for insertion sort occurs when the keys in the array are in a completely random order. For each key in a randomly-ordered array, there is an equal probability of a key at a particular index being compared to each key that comes before it, because there is an equal probability of that particular key being smaller or larger than the key that is in front (ie ½) . Mathematically, this means that for each key at index *i* in the array, there is a 1/*i* chance of it being compared to each key at index 0 through *i*-1, with 1 comparison at each index.

In other words, for a key at index i, there is a 1/*i* chance of it going through 1 key comparison, 1/*i* chance of it going through 2 key comparisons, …, 1/*i* chance of it going through i key comparisons. So, on average, a key at index *i* goes through (1/*i*)(1+2+...+*i*) = (1/*i*)( *i*\*(*i*+1)/2 ) = (*i*+1)/2 number of key comparisons.

For an array of length *n*, each key in the array, except for the first key, goes through the above process. Therefore, there are

[( (*n*-1) + 1 ) /2] + [( (*n*-2) + 1 ) /2] + … + [( 2+1) /2] + [(1+1) /2]

= (½)\* [( (*n*-1) + 1 ) + ( (*n*-2) + 1 ) + … + ( 2+1) + (1+1)]

= (½)\* [ *n* + (*n*-1) + … + 3 + 2 ]

= (½)\* [ (*n*-1)\*(*n*+2) /2 ]

= (*n*-1)\*(*n*+2) /4

number of key comparisons in an array of length n, resulting in a time complexity of O(*n*^2).

Previous answer:

Assuming the probability of going through each iteration is 1/i, the average number of comparison in the i th iteration is the sum of probability of going through each iteration(1/i) times the number of iteration (1, 2, 3, …, i). Since there will be n-1 number of iteration, it will be the total sum of each iteration from 1 to n-1, for the average number comparison in the i th iteration. This will give the time complexity of O(n^2).

Theoretical Analysis of the Time Complexity of Merge Sort

Merge sort functions by splitting the original array into 2 subarrays of equal length, and recursively splits the subarrays till each subarray contains only 1 key. When a subarray has a length of 1, it is considered trivially sorted so there is no need for any key comparison. This is the base case (W(1) = 0) for the recurrence equation which solves the time complexity of merge sort..

The comparisons of keys in the subarrays are done in the merging of a pair of subarrays that were split from an original array. This forms the recurrence equation: W(n) = 2\*W(n/2) + (number of key comparisons in the merging of subarrays of length n/2)

For an array of length *n* where *n* = 2k (k is 0 or a natural number), there will be k number of splits. This is because each split results in a subarray of length of *n*/2 = 2k-1 and every subarray is split recursively till each of them has a length of 1 = 20. k number of splits would result in k number of pairs of subarrays, which in turn results in *k* number of merges. Therefore, the number of merges in a merge sort on an array of length *n* is k = log2 *n*.

The number of key comparisons in a merge of 2 subarrays varies, depending on the case.

1. **Best Case [ (*n*/2)\*log2 *n* = O( *n*\*(lg *n*) ) ]**

The best case for merge sort occurs when every pair of subarrays being merged is such that all the keys in the first subarray are smaller than or equal to the first key of the second subarray, or all the keys in the first subarray are greater than the first key of the second subarray (and vice versa).

In the case where all the keys in the first subarray are smaller than or equal to the first key of the second subarray, each key of the first subarray will be compared against the first (smallest) key of the second subarray. After comparing and finding that the last (biggest) key of the first subarray is smaller than the first (smallest) key of the second subarray, the rest of the keys in the second subarray are assumed to be bigger than all the keys in the first subarray, and are merged without a need for comparison.

As such, the number of key comparisons in this case is the length of the first subarray. If the original array is of length *n* and the recursive splitting is such that 2 subarrays resulting from the split are of approximately equal length, then the length of the first subarray is *n*/2. This means *n*/2 number of key comparisons for each merge, so there are *n*/2 number of key comparisons.

The best case can also occur when each key at index i in the first subarray is equal to each key at index i in the second subarray, in which case both keys are merged simultaneously. When the keys from both subarrays are all merged pair-wise, simultaneously, the number of key comparisons is equal to the number of keys in either subarray (i.e. length of subarray) which is also *n*/2.

Therefore, the number of key comparisons is given by the recurrence equation:

W(*n*) = 2\*W(*n*/2) + *n*/2, or

W(2k) = 2\* W(2k-1) + 2k-1

= 2\*( 2\*W(2k-2) + 2k-2 ) + 2k-1

= 22 \* W(2k-2) + 2k-1 + 2k-1

…

= 2k \* W( 2k-k ) + k \* ( 2k-1 )

= 2k \* W(1) + k \* ( 2k-1 )

= 0 + (log2 *n*) \* *n* / 2

After simplification, this gives us a time complexity of O( *n* \* lg *n* ).

1. **Worst Case [ *n*\*log2 *n* - *n* + 1 = O( *n*\*(lg *n*) ) ]**

The worst case for merge sort occurs when the key at index *i* in the first subarray is smaller than the key at index *i* in the second subarray, but the key at index *i*+1 in the first subarray is bigger than the key at index *i* in the second subarray, for all keys in each subarray (and vice versa). That is to say, keys between the two subarrays are alternatingly bigger than preceding keys between the two subarrays.

This arrangement is such that the keys in both subarrays of length *n*/2 go through the maximum the number of comparisons in one merge: *n*/2 + *n*/2 - 1 = *n*-1. Note that the number of key comparisons is one less than the total length of the subarrays as the last elements of both subarrays are merged together.

Therefore, the number of key comparisons is given by the recurrence equation:

W(*n*) = 2\* W(*n*/2) + (*n*-1), or

W(2k) = 2\* W(2k-1) + (2k - 1)

= 2\* [ 2\* W(2k-2) + (2k-1 - 1) ] + (2k - 1)

= 22 \* W(2k-2) + (2k - 2) + (2k - 1)

= 22 \* [ 2\* W(2k-3) + (2k-2 - 1) ] + (2k - 2) + (2k - 1)

= 23 \* W(2k-3) + (2k - 22) + (2k - 21) + (2k - 20)

…

= 2k \* W(2k-k) + k \* 2k - ( 2k-1 + … + 22 + 21 + 20 )

= 2k \* W(1) + (log2 *n*) \* *n* / 2 - ( 2k -1 )

= 0 + (log2*n*) \* *n* / 2 - *n* + 1

After simplification, this gives us a time complexity of O( *n* \* lg *n* )

1. **Average Case [ (¾)\**n*\*log2 *n* - (½)\**n* + ½ = O(n\*(lg n) ]**

The average case for merge sort occurs when the keys are randomly ordered in the original array, such that when the two subarrays are merge, there is an equal probability of a key in the first array being smaller or larger than the key being compared to in the second array (ie ½). As such, keys from each subarray can be merged in any order, with equal probability. This means there is an equal probability for each possible number of key comparisons in one merge.

The least number of key comparisons that is possible in a single merge is *n*/2 (rationale given in best case) whereas the greatest number of key comparisons that is possible in a single merge is *n*-1 (rationale given in worst case). As such, the possible number of key comparisons in a single merge are: *n*/2, *n*/2 + 1, *n*/2 + 2, *n*/2 +3, …, *n*-2, *n*-1. There are *n*/2 possibilities for the number of key comparisons in one merge, and with an equal probability for each possible number of key comparisons in one merge, the probability is 1/(*n*/2) = 2/*n*.

So, on average, the number of key comparisons in a single merge is:

(2/*n*)\* [ (*n*/2) + (*n*/2 + 1) + (*n*/2 + 2) + (*n*/2 +3) + …+ (*n*-2) + (*n*-1) ]

= (2/*n*)\* [ (3*n*/2 - 1) \* (*n*/2) / 2 ]

= (3*n*/2 - 1) / 2

= (¾)\*n - ½

Therefore, the number of key comparisons is given by the recurrence equation:

W(*n*) = 2\* W(*n*/2) + 3*n*/4 - ½, or

W(2k) = 2\* W(2k-1) + (¾)\* 2k - ½

= 2\* [ 2\* W(2k-2) + (¾)\* 2k-1 - ½ ] + (¾)\* 2k - ½

= 22 \* W(2k-2) + (¾)\* 2k - 1 + (¾)\* 2k - ½

= 22 \* [ 2\* W(2k-3) + (¾)\* 2k-2 - ½ ] + (¾)\* 2k - 1 + (¾)\* 2k - ½

= 23 \* W(2k-3) + (¾)\* 2k - 2 + (¾)\* 2k - 1 + (¾)\* 2k - ½

= 23 \* W(2k-3) + 3 \* [ (¾)\* 2k ] - 21 - 20 - 2-1

…

= 2k \* W(2k-k) + k \* [ (¾)\* 2k ] - ( 2k-2 + … + 21 + 20 + 2-1)

= 2k \* W(1) + (log2 *n*) \* [ (¾)\* *n* ] - (½) \* (2k - 1) / (2-1)

= 0 + (¾)\* *n* \* log2 *n* - (½) \* (*n* - 1)

= (¾)\**n*\*log2 *n* - (½)\**n* + ½

After simplification, this gives us a time complexity of O( *n* \* lg *n* )

Conclusion