

Shubhan Singh

SE-Comps B/Batch C

2022300118

DAA Experiment 1

Aim – Experiment based on divide and conquer approach.

Details – Divide and conquer algorithm is a strategy of solving a large problem by breaking the problem into smaller sub-problems.

Input – Each student have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100,200,300,...,100000 integer numbers to Quicksort and Merge sorting algorithms.

Output –

- 1) Store the randomly generated 100000 integer numbers to a text file.
- 2) Draw a 2D plot of both sorting algorithms such that the x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the running time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.
- 3) Comment on Space complexity for two sorting algorithms.

Source code(C language):

- I have printed the output in a csv file so that the data can be easily imported into excel.

(pseudocode given later)

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define inf 0x7fffffff

#define SIZE 100000
FILE* file;

void swap(int* a, int* b){
    int temp=*b;
    *b=*a;
    *a=temp;
}

void merge(int* arr,int start, int mid, int end){
    int L[mid-start+2],R[end-mid+1];
    L[mid-start+1]=R[end-mid]=inf;
    for(int i=start;i<=mid;i++){
        L[i-start]=arr[i];
    }
    for(int i=mid+1;i<=end;i++){
        R[i-mid-1]=arr[i];
    }
}
```

```

    int i=0,l=0,r=0;
    for(int i=start;i<=end;i++){
        if(L[l]<R[r]){
            arr[i]=L[l];
            l++;
        }
        else{
            arr[i]=R[r];
            r++;
        }
    }
}

void mergesort(int* arr, int start, int end){
    if(end==start){
        return;
    }
    int mid=(end+start)/2;
    mergesort(arr,start,mid);
    mergesort(arr,mid+1,end);
    merge(arr,start,mid,end);
}

void quicksort(int* arr, int start, int end){
    if(end-start<2){
        return;
    }
}

```

```

int pivot=start;
int lower=end-1,higher=start+1;
while(1){
    while(arr[higher]<=arr[pivot]){
        higher++;
    }
    while(lower>start && arr[lower]>=arr[pivot]){
        lower--;
    }
    if(lower<higher){
        swap(&arr[pivot],&arr[lower]);
        pivot=lower;
        break;
    }
    swap(&arr[lower],&arr[higher]);
    lower--;
    higher++;
}
quicksort(arr,start,pivot);
quicksort(arr,pivot+1,end);
}

void runmergesort(int *arr, int len){
    int temp[len];
    for(int i=0;i<len;i++){
        temp[i]=arr[i];
    }
}

```

```

    clock_t time=clock();
    mergesort(temp, 0, len-1);
    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file, "%lf, ", tme);
}

void runquicksort(int* arr, int len){
    int temp[len+1];
    for(int i=0; i<len; i++){
        temp[i]=arr[i];
    }
    clock_t time=clock();
    temp[len]=inf;
    quicksort(temp, 0, len);
    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file, "%lf, ", tme);
}

void runquicksortworst(int* arr, int len){
    int temp[len+1];
    for(int i=0; i<len; i++){
        temp[i]=arr[i];
    }
    clock_t time=clock();
    temp[len]=inf;
    quicksort(temp, 0, len);

```

```

    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file,"%lf\n",tme);
}

int main(){
    if(fopen("input.txt","r")==NULL){
        srand(time(NULL));
        file=fopen("input.txt","w");
        for(int i=0;i<SIZE;i++){
            fprintf(file,"%d ",rand());
        }
        fclose(file);
    }
    file=fopen("input.txt","r");
    int arr[SIZE];
    for(int i=0;i<SIZE;i++){
        fscanf(file,"%d",&arr[i]);
    }
    int best_case_merge[SIZE];
    int worst_case_quick[SIZE];
    int worst_case_merge[SIZE];
    for(int i=0;i<SIZE;i++){
        best_case_merge[i]=i+1;
        worst_case_quick[SIZE-i-1]=i+1;
    }
}

```

```

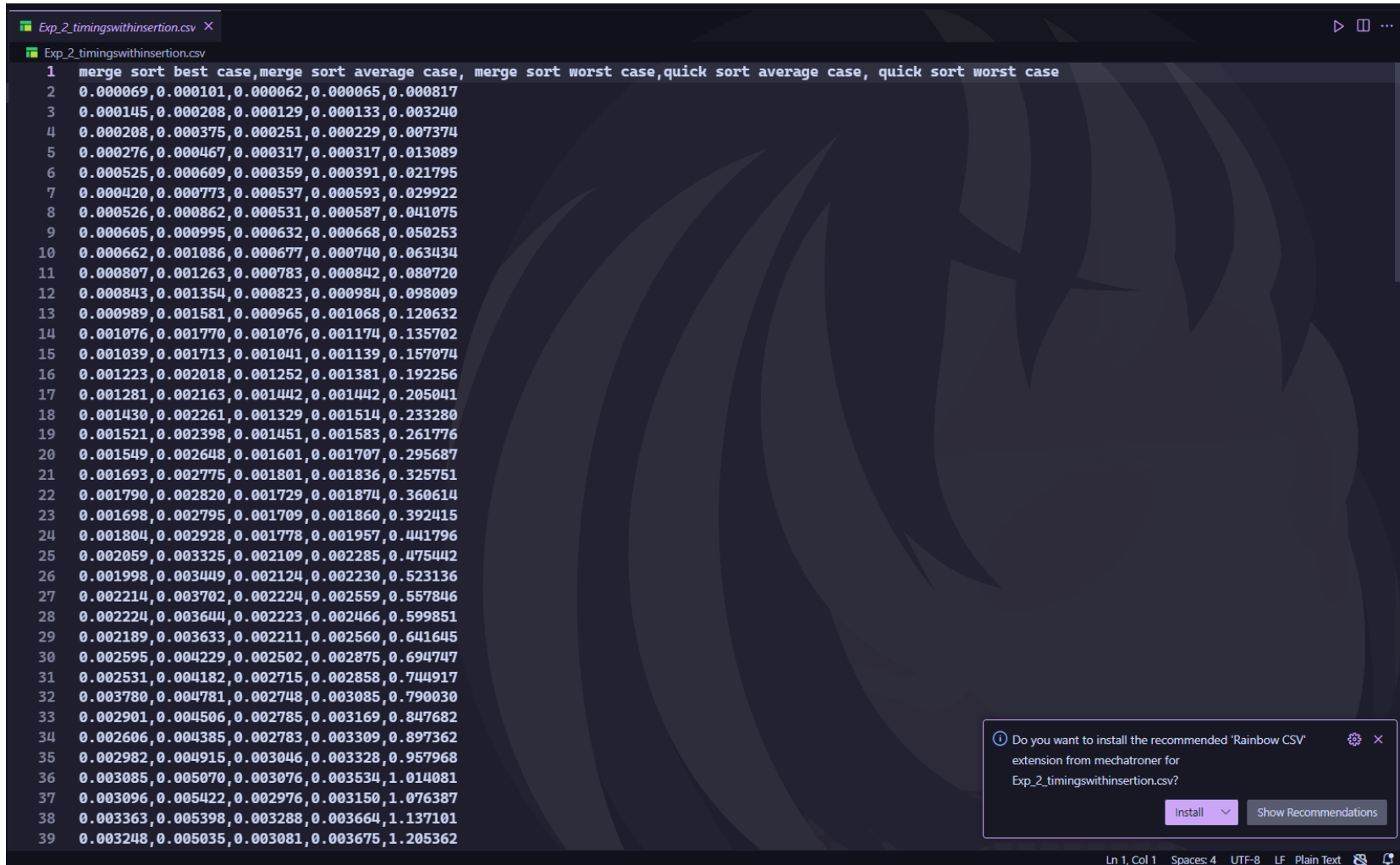
for(int i=0;i<SIZE/2;i++){
    worst_case_merge[i]=2*i+1;
}
for(int i=SIZE/2;i<SIZE;i++){
    worst_case_merge[i]=2*(i-SIZE/2+1);
}
fclose(file);
file=fopen("Exp_2_timings.csv","w");
fprintf(file,"merge sort best case,merge sort average case, merge sort worst case,quick sort
average case, quick sort worst case\n");
for(int i=1000;i<=SIZE;i+=1000){
    runmergesort(best_case_merge,i);
    runmergesort(arr,i);
    runmergesort(worst_case_merge,i);
    runquicksort(arr,i);
    runquicksortworst(worst_case_quick,i);
}
fclose(file);

return 0;
}

```

Output:

File created



```
1 merge sort best case,merge sort average case, merge sort worst case,quick sort average case, quick sort worst case
2 0.000069,0.000101,0.000062,0.000065,0.000817
3 0.000145,0.000208,0.000129,0.000133,0.003240
4 0.000208,0.000375,0.000251,0.000229,0.007374
5 0.000276,0.000467,0.000317,0.000317,0.013089
6 0.000525,0.000609,0.000359,0.000391,0.021795
7 0.000420,0.000773,0.000537,0.000593,0.029922
8 0.000526,0.000862,0.000531,0.000587,0.041075
9 0.000605,0.000995,0.000632,0.000668,0.050253
10 0.000662,0.001086,0.000677,0.000740,0.063434
11 0.000807,0.001263,0.000783,0.000842,0.080720
12 0.000843,0.001354,0.000823,0.000984,0.098009
13 0.000989,0.001581,0.000965,0.001068,0.120632
14 0.001076,0.001770,0.001076,0.001174,0.135702
15 0.001039,0.001713,0.001041,0.001139,0.157074
16 0.001223,0.002018,0.001252,0.001381,0.192256
17 0.001281,0.002163,0.001442,0.001442,0.205041
18 0.001430,0.002261,0.001329,0.001514,0.233280
19 0.001521,0.002398,0.001451,0.001583,0.261776
20 0.001549,0.002648,0.001601,0.001707,0.295687
21 0.001693,0.002775,0.001801,0.001836,0.325751
22 0.001790,0.002820,0.001729,0.001874,0.360614
23 0.001698,0.002795,0.001709,0.001860,0.392415
24 0.001804,0.002928,0.001778,0.001957,0.441796
25 0.002059,0.003325,0.002109,0.002285,0.475442
26 0.001998,0.003449,0.002124,0.002230,0.523136
27 0.002214,0.003702,0.002224,0.002559,0.557846
28 0.002224,0.003644,0.002223,0.002466,0.599851
29 0.002189,0.003633,0.002211,0.002560,0.641645
30 0.002595,0.004229,0.002502,0.002875,0.694747
31 0.002531,0.004182,0.002715,0.002858,0.744917
32 0.003780,0.004781,0.002748,0.003085,0.790030
33 0.002901,0.004506,0.002785,0.003169,0.847682
34 0.002606,0.004385,0.002783,0.003309,0.897362
35 0.002982,0.004915,0.003046,0.003328,0.957968
36 0.003085,0.005070,0.003076,0.003534,1.014081
37 0.003096,0.005422,0.002976,0.003150,1.076387
38 0.003363,0.005398,0.003288,0.003664,1.137101
39 0.003248,0.005035,0.003081,0.003675,1.205362
```

Do you want to install the recommended 'Rainbow CSV' extension from mechatroner for Exp_2_timingswithinsertion.csv?

Install Show Recommendations

Ln 1, Col 1 Spaces: 4 UTF-8 LF Plain Text

AutoSave Exp_2_timings Search Shubhan Singh

File Home Insert Page Layout Formulas Data Review View Automate Help

Clipboard Font Alignment Number Styles Cells Editing Add-ins Analyse Data

A1 merge sort best case

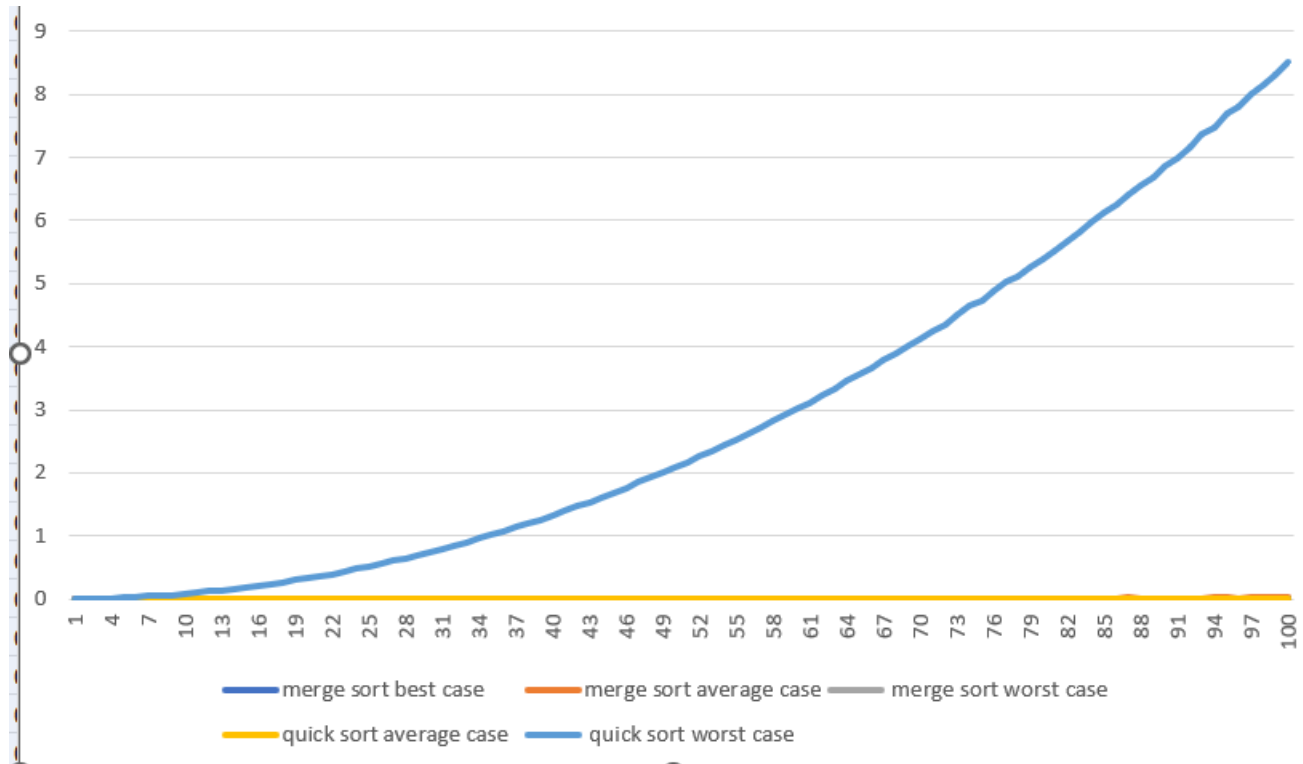
	A	B	C	D	E	F	G	H	I	J	K	L	M
1	merge sort best case	merge sort average case	merge sort worst case	quick sort average case	quick sort worst case								
2	0.000069	0.000101	0.000062	0.000065	0.000817								
3	0.000145	0.000208	0.000129	0.000133	0.00324								
4	0.000208	0.000375	0.000251	0.000229	0.007374								
5	0.000276	0.000467	0.000317	0.000317	0.013089								
6	0.000525	0.000609	0.000359	0.000391	0.021795								
7	0.00042	0.000773	0.000537	0.000593	0.029922								
8	0.000526	0.000862	0.000531	0.000587	0.041075								
9	0.000605	0.000995	0.000632	0.000668	0.050253								
10	0.000662	0.001086	0.000677	0.00074	0.063434								
11	0.000807	0.001263	0.000783	0.000842	0.08072								
12	0.000843	0.001354	0.000823	0.000984	0.098009								
13	0.000989	0.001581	0.000965	0.001068	0.120632								
14	0.001076	0.00177	0.001076	0.001174	0.135702								
15	0.001039	0.001713	0.001041	0.001139	0.157074								
16	0.001223	0.002018	0.001252	0.001381	0.192256								
17	0.001281	0.002163	0.001442	0.001442	0.205041								
18	0.00143	0.002261	0.001329	0.001514	0.23328								
19	0.001521	0.002398	0.001451	0.001583	0.261776								
20	0.001549	0.002648	0.001601	0.001707	0.295687								
21	0.001693	0.002775	0.001801	0.001836	0.325751								
22	0.00179	0.00282	0.001729	0.001874	0.360614								
23	0.001698	0.002795	0.001709	0.00186	0.392415								
24	0.001804	0.002928	0.001778	0.001957	0.441796								
25	0.002000	0.003235	0.002000	0.002000	0.475113								

Exp_2_timings

Ready Accessibility: Unavailable 100%

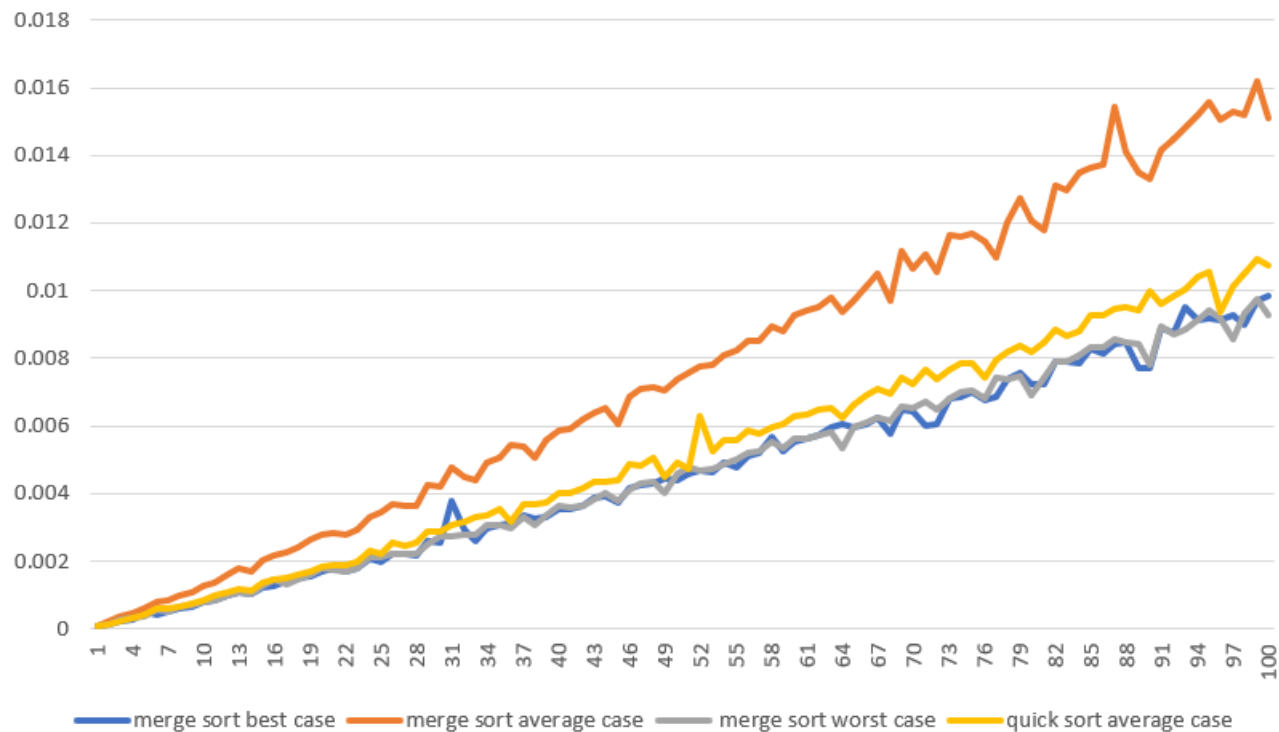
CSV file

PLOTS:



Graph of time taken by both algorithms to sort 100 blocks of 1000,2000...100000 elements.

The worst case of Quicksort is polynomially slower than the other cases, thus the others just appear as a flat line.



The other graphs, apart from worst case of quicksort.

All are of $O(n \log n)$ time complexity.

Conclusion:

- Since the graphs of the running times average cases of the sorting algorithms with respect to the input size are similar to graphs of function $n \log n$, it can be said that insertion and selection sort have $O(n \log n)$ average case time complexity.
- Merge sort has $O(n \log n)$ best, worse and average case time complexity.
- Quicksort has $O(n^2)$ time complexity.

Pseudocode:

1. Merge function:

Create two temporary arrays L and R

Copy the elements from the original array to these temporary arrays

Initialize two pointers l and r to the start of L and R respectively

For each element from start to end in the original array

 If the current element in L is less than the current element in R

 Assign the current element in L to the original array

 Increment the pointer l

 Else

 Assign the current element in R to the original array

 Increment the pointer r

2. Mergesort function:

If the start index is not equal to the end index (i.e., the array has more than one element)

 Calculate the mid index

 Recursively call mergesort for the left half of the array

 Recursively call mergesort for the right half of the array

 Merge the two halves using the merge function

3. Quicksort function:

If the array has more than one element

Choose the first element as the pivot

Initialize two pointers, lower and higher, to the start and end of the array respectively

While the lower pointer is less than the higher pointer

Increment the higher pointer while the current element is less than or equal to the pivot

Decrement the lower pointer while the current element is greater than or equal to the pivot

If the lower pointer is still less than the higher pointer

Swap the elements at the lower and higher pointers

Else

Swap the pivot with the element at the lower pointer and update the pivot index

Recursively call quicksort for the two halves divided by the pivot

Theory:

Quicksort– It picks an element called as pivot, and then it partitions the given array around the picked pivot element. It then arranges the entire array in two sub-array such that one array holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot. The Divide and Conquer steps of Quicksort perform following functions.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort

Combine: Combine the already sorted array.

Merge sort– Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging. The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two element lists, sorting them in the process. The sorted two-element pairs are merged into the four-element lists, and so on until we get the sorted list.

Time and space complexities of insertion and selection sort:

Merge Sort:

Time Complexity:

- Best, Average, and Worst Case: $O(n \log n)$

Space Complexity: $O(n)$ as it requires temporary arrays during the merge step

Here's a simple visualisation of the process behind merge sort:

Merge sort:

$n=4$ $[3, 2, 1, 4]$

Split \rightarrow $[3, 2]$ $[1, 4]$
split \rightarrow $[3]$ $[2]$ $[1]$ $[4]$ $\left. \vphantom{\begin{array}{l} \text{Split} \\ \text{split} \end{array}} \right\} \log_2 n \text{ steps to split}$

Merge \rightarrow $[2, 3]$ $[1, 4]$ $\left. \vphantom{\begin{array}{l} \text{Split} \\ \text{split} \end{array}} \right\} O(n) \text{ merge}$

Merge \rightarrow $[1, 2, 3, 4]$

Quick Sort:

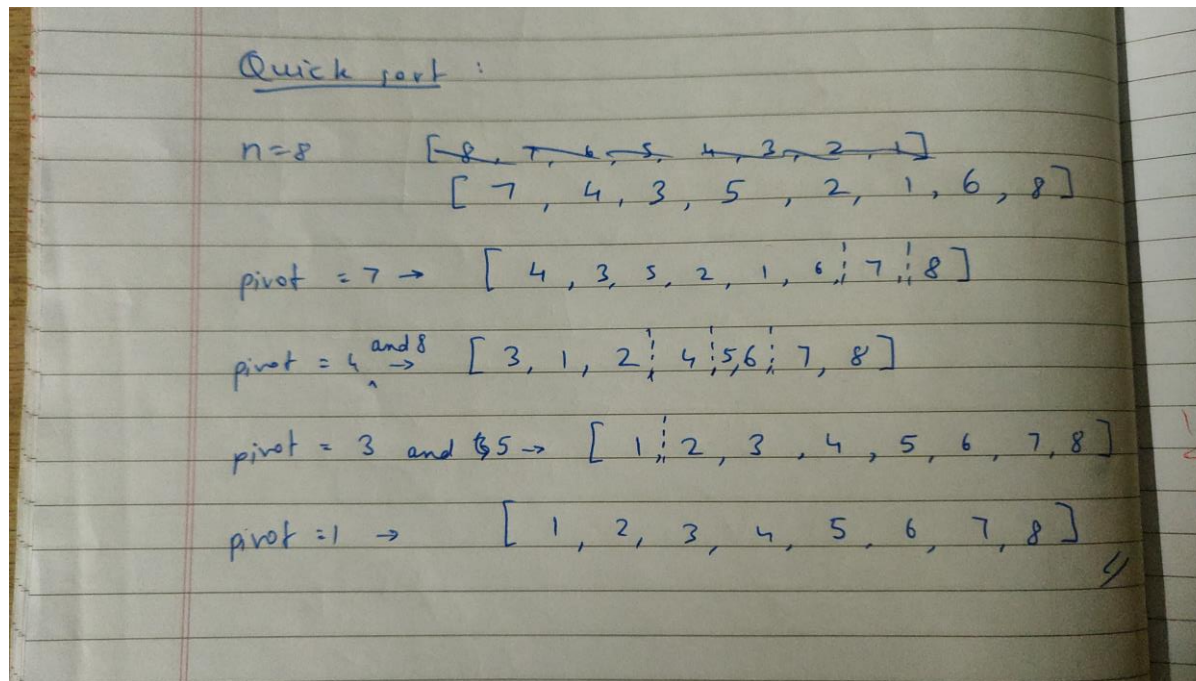
Time Complexity:

- Best and Average Case: $O(n \log n)$ when the pivot element is near the median on average
- Worst Case: $O(n^2)$ when the pivot element is the smallest or largest element (e.g., in a sorted or reverse sorted array)
- The worst case can be avoided by using methods like the median of three method to choose the pivot.

Space Complexity: $O(\log n)$ for the recursive stack space. It's an in-place sorting algorithm, but recursive calls add to the space complexity.

But it does not require extra arrays.

Here is a simple visualisation of quick sort:



As can be seen, for both algorithms, the number of levels of splitting, merging, or partitioning (which is proportional to the time complexity) increases logarithmically with the size of the input, hence the $O(n \log n)$ time complexity for both algorithms in the average case. The space complexity for Merge Sort is $O(n)$ because of the temporary arrays used during merging, while Quick Sort has a space complexity of $O(\log n)$ due to the recursive stack space.