Name: Vishal Shashikant Salvi.
UID: 2019230069
Batch: C
Class: SE Comps

Experiment No 2

Aim: Experiment Based on Divide and conquer approach: Quick sort.

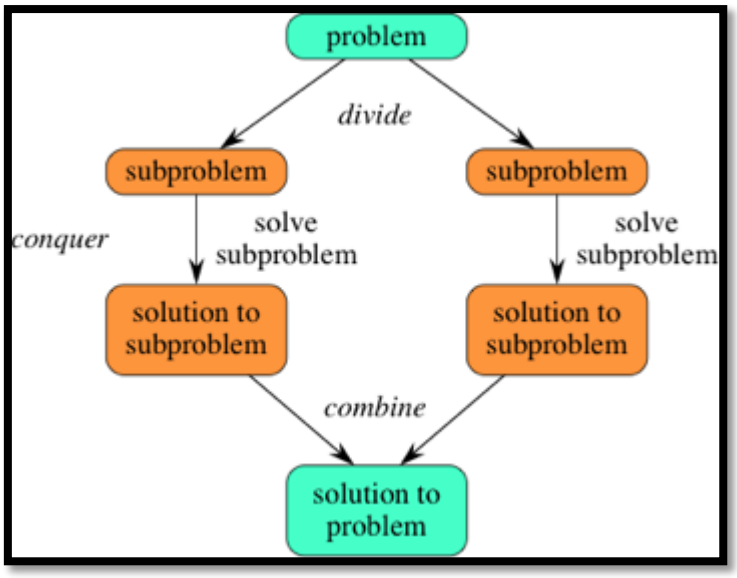
Theory:

Divide-and-conquer

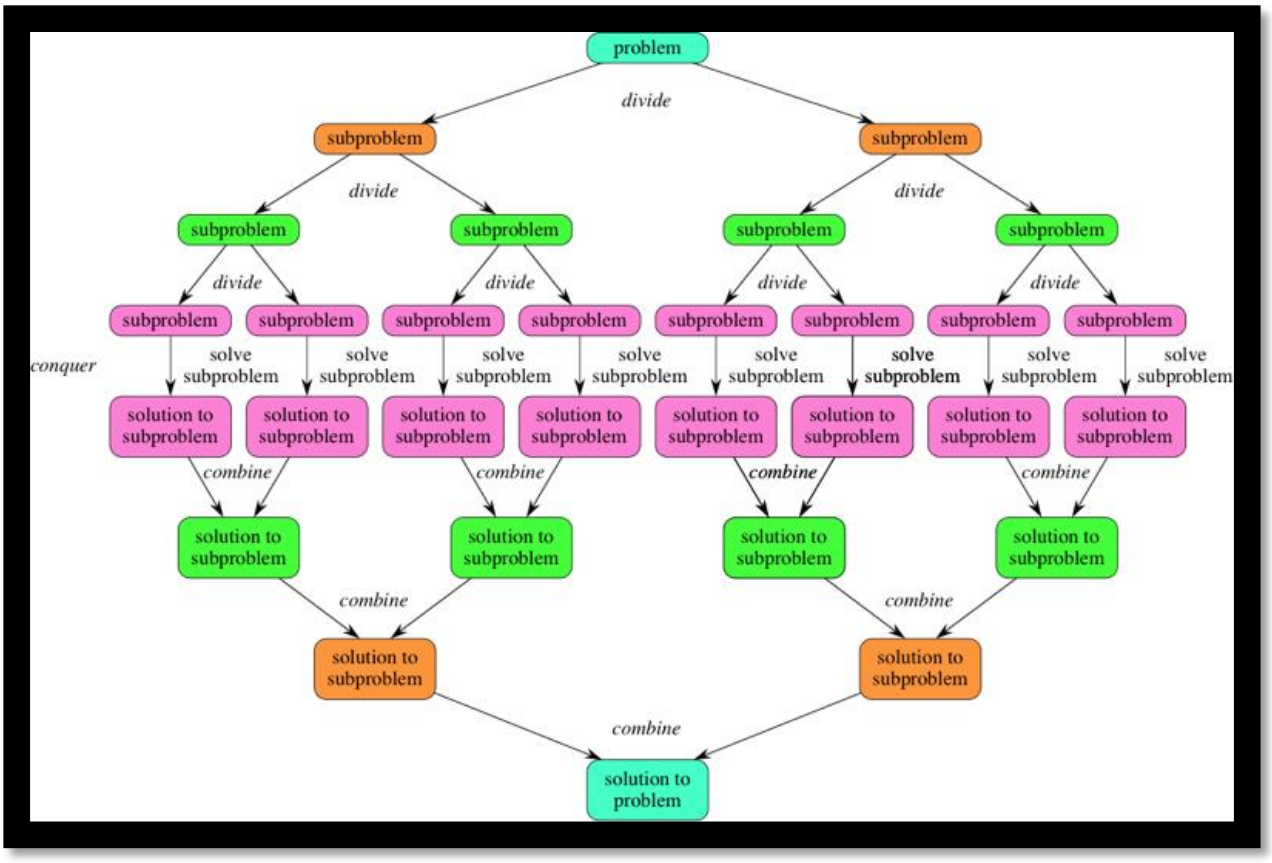
Both merge sort and quicksort employ a common algorithmic paradigm based on recursion. This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as *divide, conquer, combine*. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):



If we expand out two more recursive steps, it looks like this:



Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.

Quick Sort:

Like merge sort, quicksort uses divide-and-conquer, and so it's a recursive algorithm. The way that quicksort uses divide-and-conquer is a little different from how merge sort does. In merge sort, the divide step does hardly anything, and all the real work happens in the combine step. Quicksort is the opposite: all the real work happens in the divide step. In fact, the combine step in quicksort does absolutely nothing.

Quicksort has a couple of other differences from merge sort. Quicksort works in place. And its worst-case running time is as bad as selection sort's and insertion sort's: \Theta(n^2)Θ(*n*2)\Theta, left parenthesis, n, squared, right parenthesis. But its average-case running time is as good as merge sort's: \Theta(n \log_2 n)Θ(*n*log2*n*)\Theta, left parenthesis, n, log, start base, 2, end base, n, right parenthesis. So why think about quicksort when merge sort is at least as good? That's because the constant factor hidden in the big-Θ notation for quicksort is quite good. In practice, quicksort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

Here is how quicksort uses divide-and-conquer. As with merge sort, think of sorting a subarray array[p..r], where initially the subarray is array[0..n-1].

1. **Divide** by choosing any element in the subarray array[p..r]. Call this element the **pivot**. Rearrange the elements in array[p..r] so that all elements in array[p..r] that are less than or equal to the pivot are to its left and all elements that are greater than the pivot are to its right. We call this procedure **partitioning**. At this point, it doesn't matter what order the elements to the left of the pivot are in relation to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.

   As a matter of practice, we'll always choose the rightmost element in the subarray, array[r], as the pivot. So, for example, if the subarray consists of [9, 7, 5, 11, 12, 2, 14, 3, 10, 6], then we choose 6 as the pivot. After partitioning, the subarray might look like [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]. Let q be the index of where the pivot ends up.

2. **Conquer** by recursively sorting the subarrays array[p..q-1] (all elements to the left of the pivot, which must be less than or equal to the pivot) and array[q+1..r] (all elements to the right of the pivot, which must be greater than the pivot).

3. **Combine** by doing nothing. Once the conquer step recursively sorts, we are done. Why? All elements to the left of the pivot, in array[p..q-1], are less than or equal to the pivot and are sorted, and all elements to the right of the pivot, in array[q+1..r], are greater than the pivot and are sorted. The elements in array[p..r] can't help but be sorted!

   Think about our example. After recursively sorting the subarrays to the left and right of the pivot, the subarray to the left of the pivot is [2, 3, 5], and the subarray to the right of the pivot is [7, 9, 10, 11, 12, 14]. So the subarray has [2, 3, 5], followed by 6, followed by [7, 9, 10, 11, 12, 14]. The subarray is sorted.

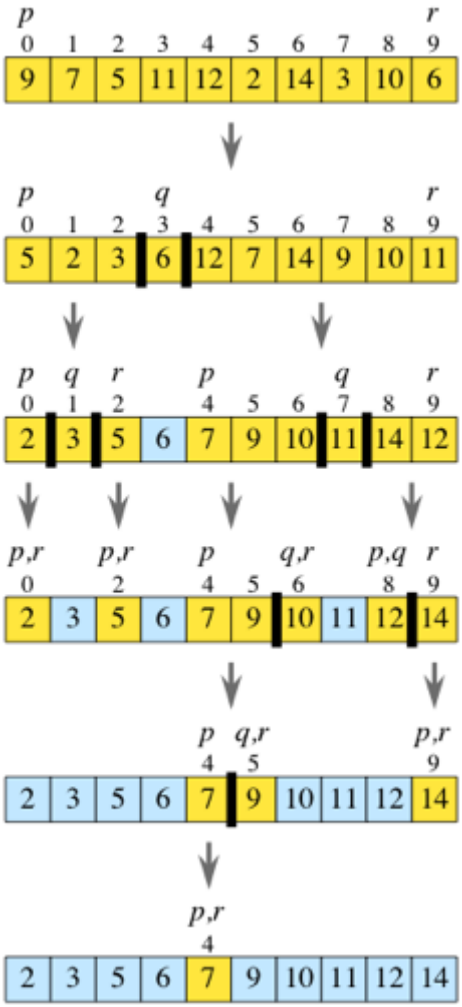
   The base cases are subarrays of fewer than two elements, just as in merge sort. In merge sort, you never see a subarray with no elements, but you can in quicksort, if the other elements in the subarray are all less than the pivot or all greater than the pivot.

   Let's go back to the conquer step and walk through the recursive sorting of the subarrays. After the first partition, we have subarrays of [5, 2, 3] and [12, 7, 14, 9, 10, 11], with 6 as the pivot.

   To sort the subarray [5, 2, 3], we choose 3 as the pivot. After partitioning, we have [2, 3, 5]. The subarray [2], to the left of the pivot, is a base case when we recurse, as is the subarray [5], to the right of the pivot.

   To sort the subarray [12, 7, 14, 9, 10, 11], we choose 11 as the pivot, resulting in [7, 9, 10] to the left of the pivot and [14, 12] to the right. After these subarrays are sorted, we have [7, 9, 10], followed by 11, followed by [12, 14].

   Here is how the entire quicksort algorithm unfolds. Array locations in blue have been pivots in previous recursive calls, and so the values in these locations will not be examined or moved again:



Algorithm:

Quick-Sort (A, p, r)

```
if p < r then
  q Partition (A, p, r)
  Quick-Sort (A, p, q)
  Quick-Sort (A, q + r, r)
```

Partition (A, p, r)

```
x ← A[p]
i ← p-1
j ← r+1
while TRUE do
```

```
Repeat j ← j - 1
until A[j] ≤ x
Repeat i← i+1
until A[i] ≥ x
if i < j then
  exchange A[i] ↔ A[j]
else
  return j
```

Time Complexity:

- **Worst Case Complexity [Big-O]**: $O(n^2)$

  It occurs when the pivot element picked is always either the greatest or the smallest element.

  In the above algorithm, if the array is in descending order, the partition algorithm always picks the smallest element as a pivot element.

- **Best Case Complexity [Big-omega]**: $O(n*\log n)$

  It occurs when the pivot element is always the middle element or near to the middle element.

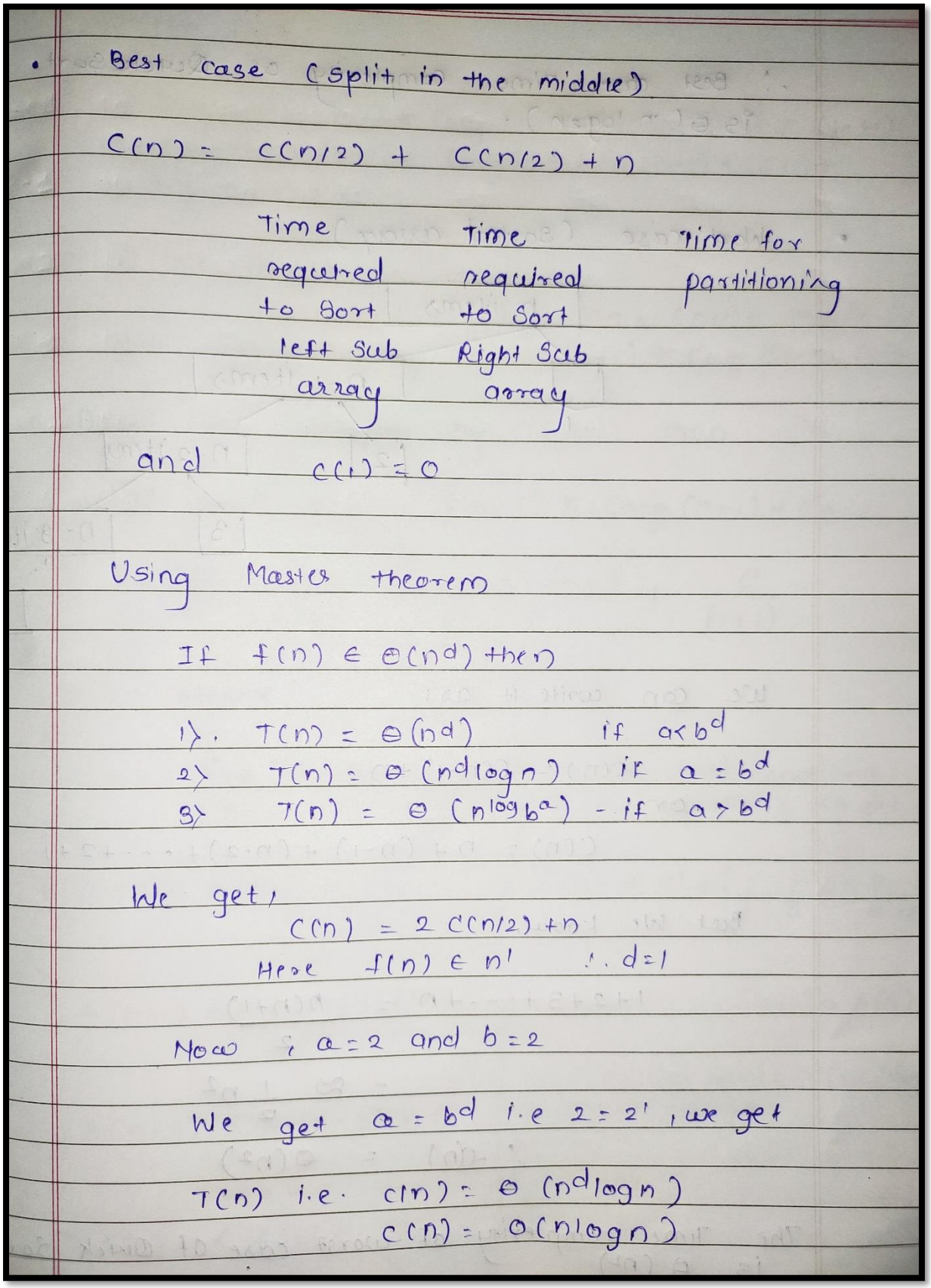
- **Average Case Complexity [Big-theta]**: $O(n*\log n)$

  It occurs when the above conditions do not occur.

Space Complexity

The space complexity for quicksort is $O(n*\log n)$.

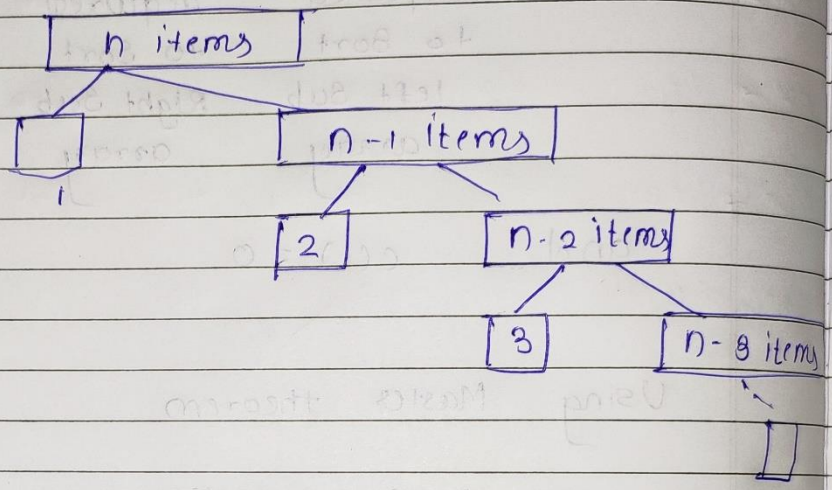
Best Case:


Best case (split in the middle)
C(n) = C(n/2) + C(n/2) + n
Time required to Sort left Sub array
Time required to Sort Right Sub array
Time for partitioning
and C(1) = 0
Using Master theorem
If f(n) ∈ Θ(nd) then
1). T(n) = Θ(nd) if a < bd
2) T(n) = Θ (ndlogn) if a = bd
3) T(n) = Θ (n^log b^a) - if a > bd
We get,
C(n) = 2 C(n/2) + n
Here f(n) ∈ n^1 ∴ d=1
Now, a=2 and b=2
We get a = bd i.e 2 = 2^1, we get
T(n) i.e. c(n) = Θ (ndlogn)
c(n) = O(nlogn)

Worst Case:

∴ Best case Time complexity of Quick Sort is $\Theta(n \log_2 n)$.

- Worst case (sorted array)



We can write it as:

$$C(n) = C(n-1) + n$$

or

$$C(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$

But we know,

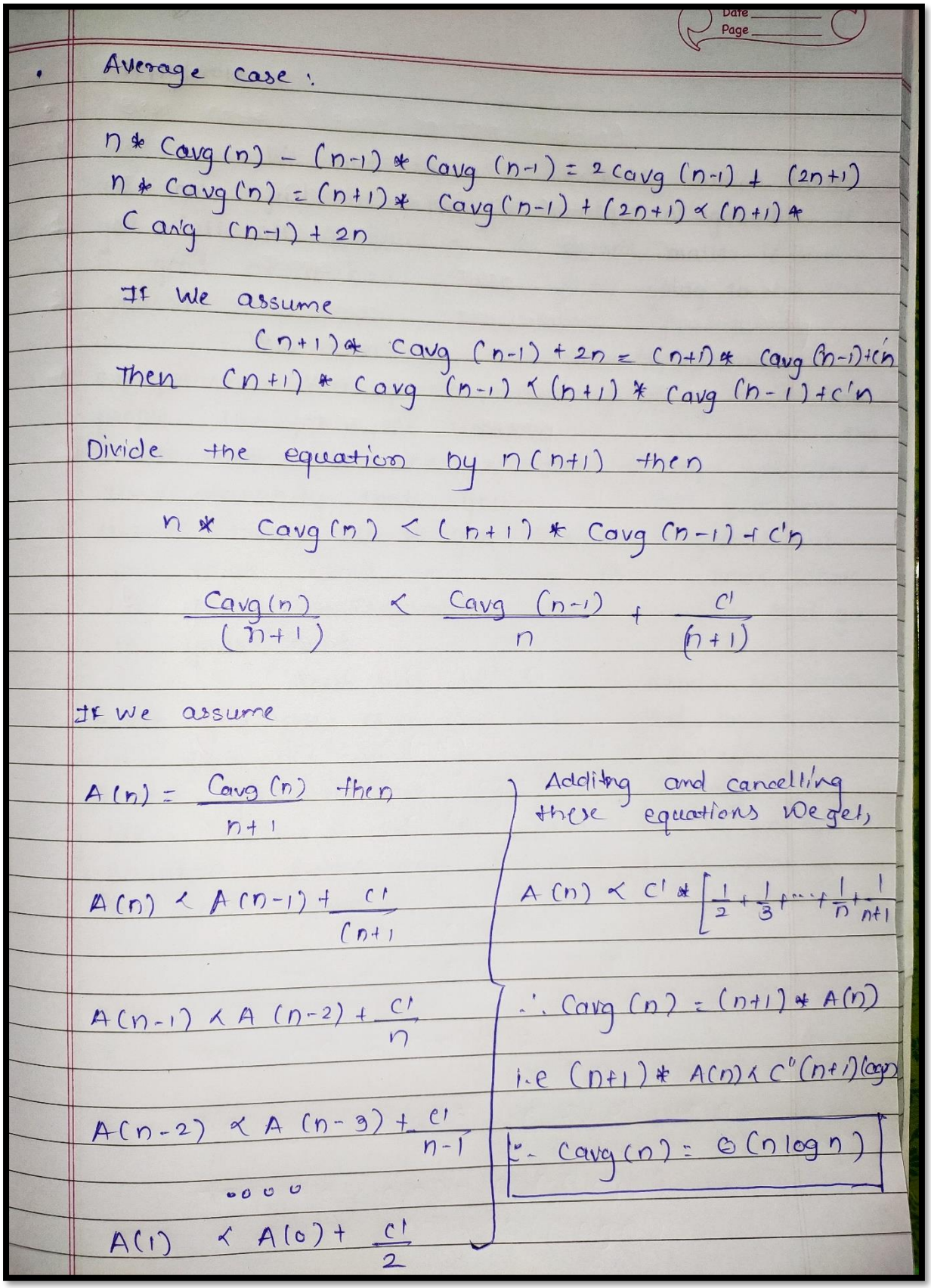
$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$= \approx \frac{1}{2} n^2$$

$$\therefore C(n) = \Theta(n^2)$$

The Time complexity of worst case of Quick Sort is $\Theta(n^2)$

Average Case:



Average case :

$n * Cavg(n) - (n-1) * Cavg(n-1) = 2\,Cavg(n-1) + (2n+1)$

$n * Cavg(n) = (n+1) * Cavg(n-1) + (2n+1) \propto (n+1) * Cavg(n-1) + 2n$

If we assume

$(n+1) * Cavg(n-1) + 2n = (n+1) * Cavg(n-1) + c'n$

Then $(n+1) * Cavg(n-1) \propto (n+1) * Cavg(n-1) + c'n$

Divide the equation by $n(n+1)$ then

$$n * Cavg(n) \propto (n+1) * Cavg(n-1) + c'n$$

$$\frac{Cavg(n)}{(n+1)} \propto \frac{Cavg(n-1)}{n} + \frac{c'}{(n+1)}$$

If we assume

$A(n) = \frac{Cavg(n)}{n+1}$ then

$A(n) \propto A(n-1) + \frac{c'}{(n+1)}$

$A(n-1) \propto A(n-2) + \frac{c'}{n}$

$A(n-2) \propto A(n-3) + \frac{c'}{n-1}$

....

$A(1) \propto A(0) + \frac{c'}{2}$

Adding and cancelling these equations we get,

$A(n) \propto c' * \left[\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \frac{1}{n+1}\right]$

$\therefore Cavg(n) = (n+1) * A(n)$

i.e $(n+1) * A(n) \propto c''(n+1)\log n$

$\therefore Cavg(n) = \Theta(n \log n)$

Analysis:

Worst-case running time

When quicksort always has the most unbalanced partitions possible, then the original call takes cn*cn*c, n time for some constant c*c*c, the recursive call on n-1*n*−1n, minus, 1 elements takes c(n-1)*c*(*n*−1)c, left parenthesis, n, minus, 1, right parenthesis time, the recursive call on n-2*n*−2n, minus, 2 elements takes c(n-2)*c*(*n*−2)c, left parenthesis, n, minus, 2, right parenthesis time, and so on. Here's a tree of the subproblem sizes with their partitioning times:

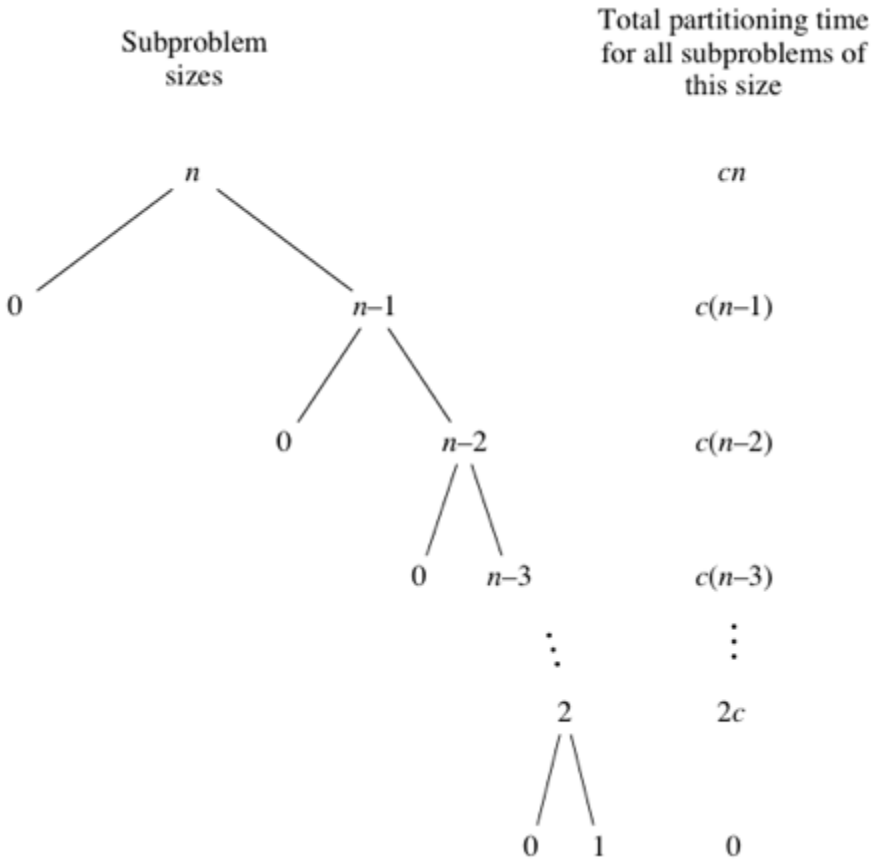


Diagram of worst case performance for Quick Sort, with a tree on the left and partition times on the right. The tree is labeled "Subproblem sizes" and the right is labeled "Total partitioning time for all subproblems of this size." The first level of the tree shows a single node n and corresponding partitioning time of c times n. The second level of the tree shows two nodes, 0 and n minus 1, and a partitioning time of c times n minus 1. The third level of the tree shows two nodes, 0 and n minus 2, and a partitioning time of c times n minus 2. The fourth level of

the tree shows two nodes, 0 and n minus 3, and a partitioning time of c times n minus 3. Underneath that level, dots indicate that the tree continues like that. The second to last level in the tree has a single node 2 with a partitioning time of 2 times c and the last level has two nodes of 0 and 1, with a partitioning time of 0.

When we total up the partitioning times for each level, we get

$$cn + c(n-1) + c(n-2) + \cdots + 2c = c(n + (n-1) + (n-2) + \cdots$$
$$= c((n+1)(n/2) - 1) .$$
$$= \Theta(n^2).$$

Best-case running time

Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2(*n*−1)/2left parenthesis, n, minus, 1, right parenthesis, slash, 2 elements. The latter case occurs if the subarray has an even number n of elements and one partition has n/2*n*/2n, slash, 2 elements with the other having n/2-1*n*/2−1n, slash, 2, minus, 1. In either of these cases, each partition has at most n/2*n*/2n, slash, 2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:

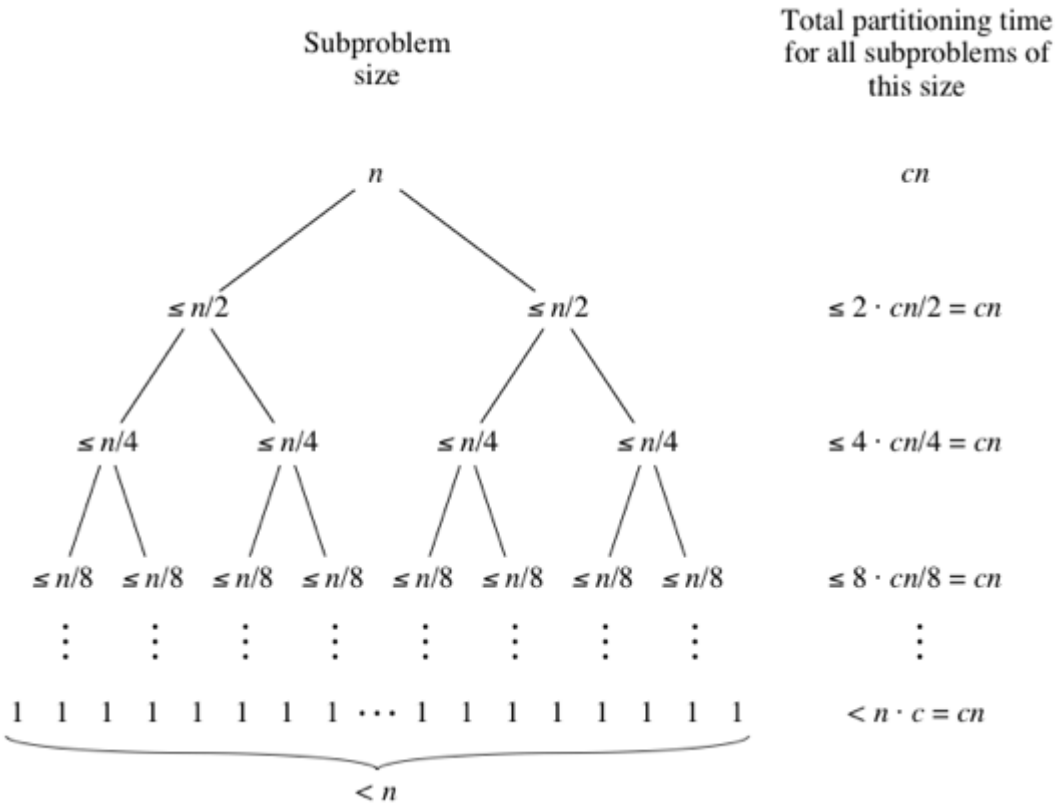


Diagram of best case performance for Quick Sort, with a tree on the left and partitioning times on the right. The tree is labeled "Subproblem size" and the right is labeled "Total partitioning time for all subproblems of this size." The first level of the tree shows a single node n and corresponding partitioning time of c times n. The second level of the tree shows two nodes, each of less than or equal to 1/2 n, and a partitioning time less than or equal to 2 times c times 1/2 n, the same as c times n. The third level of the tree shows four nodes, each of less than or equal to 1/4 n, and a partitioning time less than or equal to 4 times c times 1/4 n, the same as c times n. The fourth level of the tree shows eight nodes, each of less than ot equal to 1/8 n, and a partitioning time less than or equal to 8 times c times 1/8 n, the same as c times n. Underneath that level, dots are shown to indicate the tree continues like that. A final level is shown with n nodes of 1, and a partitioning time of less than or equal to n times c, the same as c times n.

Using big-Θ notation, we get the same result as for merge sort: $\Theta(n \log_2 n)$.

Average-case running time

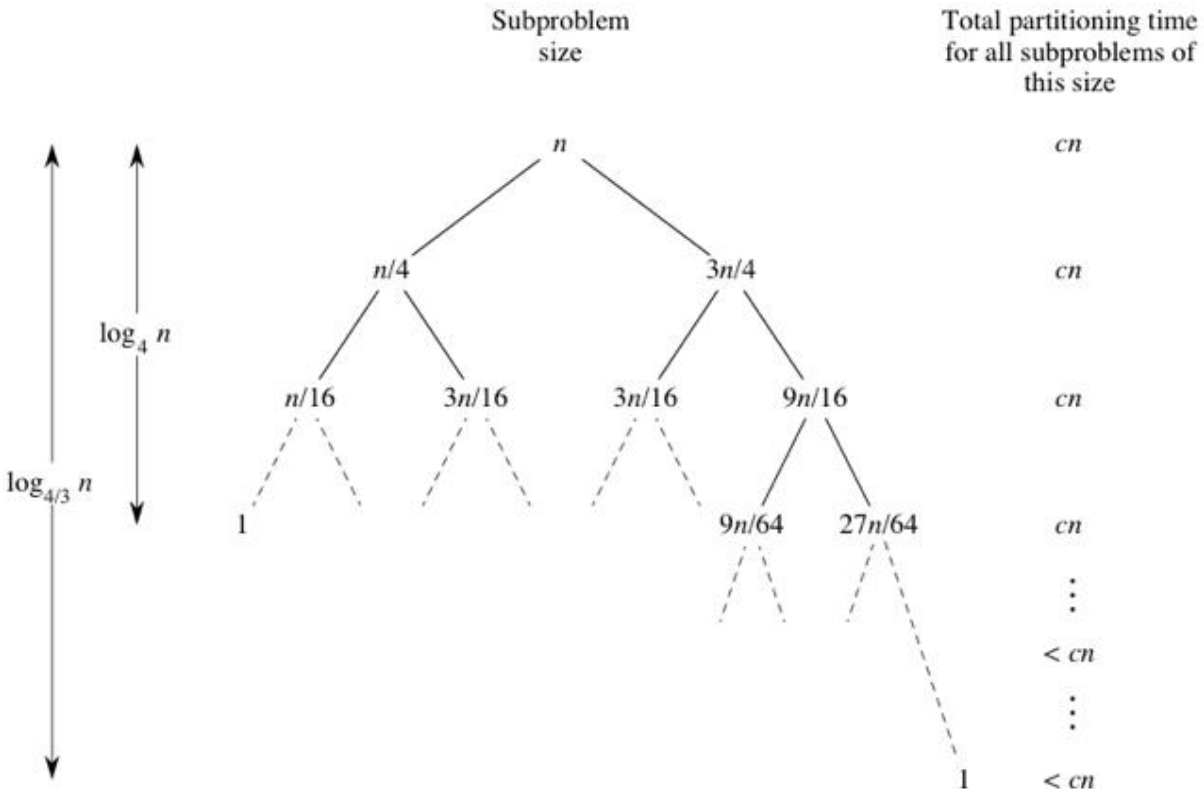
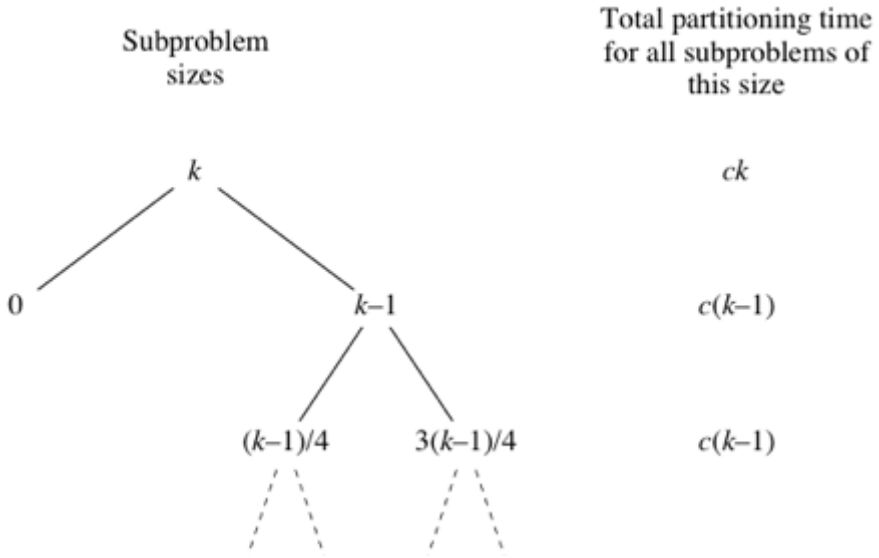
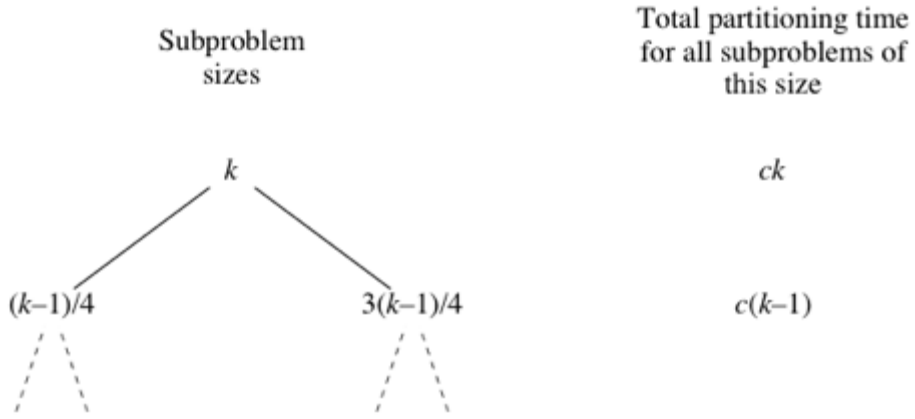


Diagram of average case performance for Quick Sort

The other case we'll look at to understand why quicksort's average-case running time is O(n \log_2 n)$O(n\log_2 n)$O, left parenthesis, n, log, start base, 2, end base, n, right parenthesis is what would happen if the half of the time that we don't get a 3-to-1 split, we got the worst-case split. Let's suppose that the 3-to-1 and worst-case splits alternate, and think of a node in the tree with kkk elements in its subarray. Then we'd see a part of the tree that looks like this:



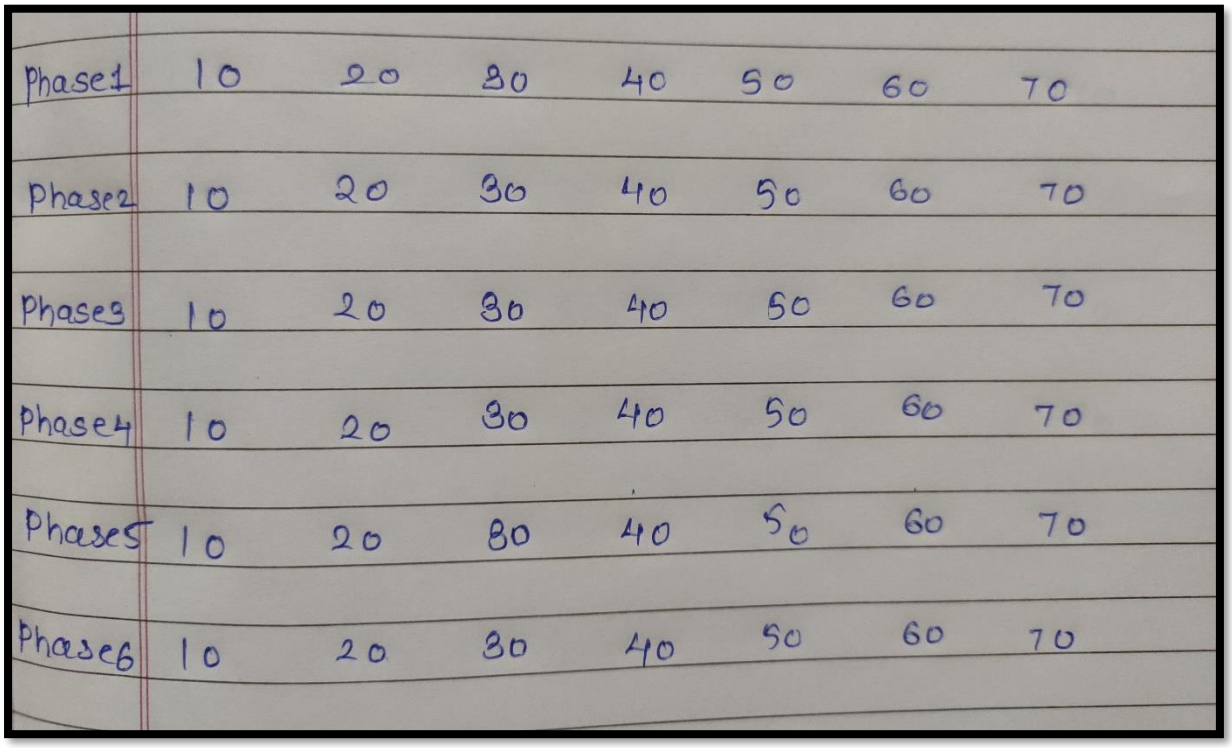
instead of like this:



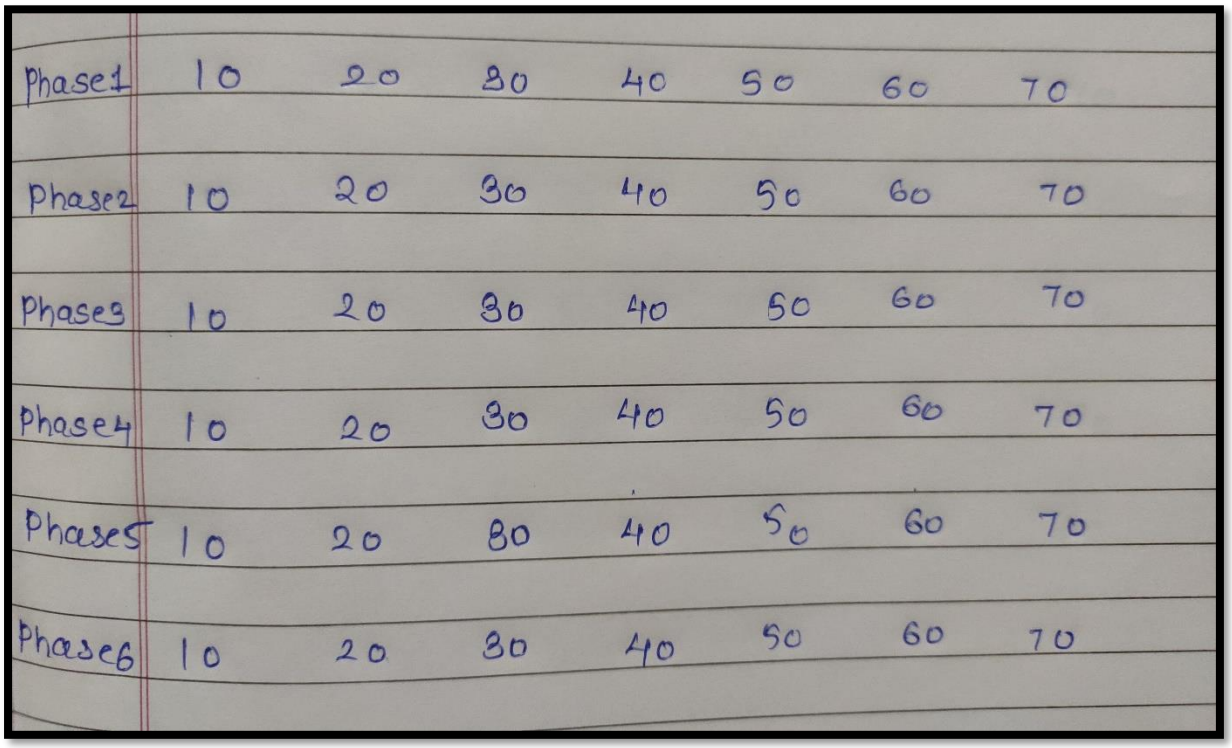
Therefore, even if we got the worst-case split half the time and a split that's 3-to-1 or better half the time, the running time would be about twice the running time of getting a 3-to-1 split every time. Again, that's just a constant factor, and it gets absorbed into the big-O notation, and so in this case, where we alternate between worst-case and 3-to-1 splits, the running time is O(n log_2 n)$O(n\log 2n)$O, left parenthesis, n, log, start base, 2, end base, n, right parenthesis.

Bear in mind that this analysis is *not* mathematically rigorous, but it gives you an intuitive idea of why the average-case running time might be O(n log_2 n)$O(n\log 2n)$O, left parenthesis, n, log, start base, 2, end base, n, right parenthesis.

Best Case:

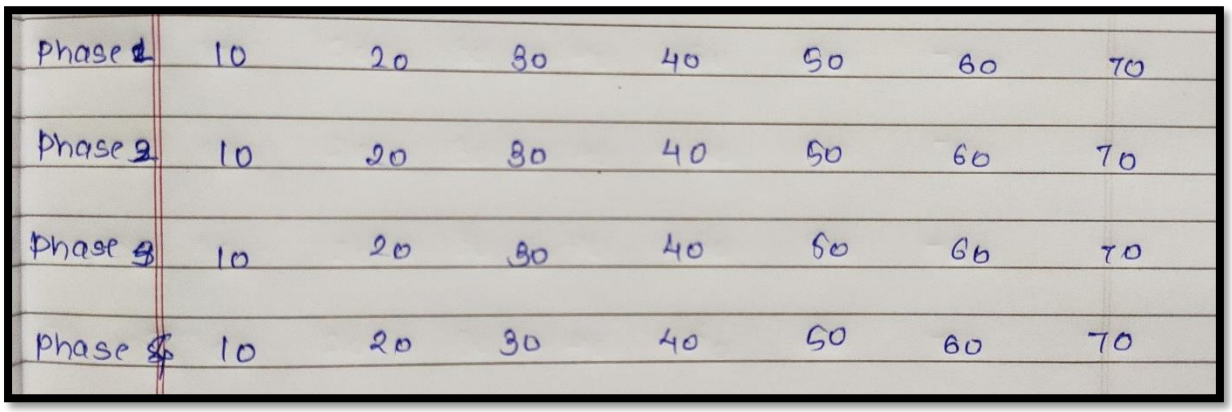


Worst case:



Phase1	10	20	30	40	50	60	70
Phase2	10	20	30	40	50	60	70
Phase3	10	20	30	40	50	60	70
Phase4	10	20	30	40	50	60	70
Phase5	10	20	30	40	50	60	70
Phase6	10	20	30	40	50	60	70

Average Case:



Phase1	10	20	30	40	50	60	70
Phase2	10	20	30	40	50	60	70
Phase3	10	20	30	40	50	60	70
Phase4	10	20	30	40	50	60	70

Program Code:

```
#include <stdio.h>


int a[20];
int size,i;

void printArr()
{
printf("\nNew Phase:\n");
for(int i=0;i<size;i++)
{
printf("%d ",a[i]);

}
}


void quick_sort(int[],int,int);
int partition(int[],int,int);

int count1=0,count2=0;

int main()
{


        printf("Enter the number of elements: ");
        scanf("%d",&size);
        printf("\nEnter array elements:");

        for(i=0;i<size;i++)
                scanf("%d",&a[i]);



        quick_sort(a,0,size-1);
```

```
	printf("\nTotal no. of comparisons: %d",count1);
	printf("\nTotal no. of swaps: %d",count2);


}

void quick_sort(int a[],int l,int u)
{
	int j;
	if(l<u)
	{
		j=partition(a,l,u);
		quick_sort(a,l,j-1);
		quick_sort(a,j+1,u);
		printArr(a[i]);

	}


}

int partition(int a[],int l,int u)
{
	int v,j,temp;
	v=a[l];
	i=l;
	j=u+1;

	do
	{
	count2++;
		do{
	      count1++;
	     i++;
	   }while(a[i]<v && i<=u);

		do{
	      count1++;
```

```
            j--;
          }while(v < a[j]);

              if(i<j)
              {
                    temp=a[i];
                    a[i]=a[j];
                    a[j]=temp;

              }
        }while(i<j);

        a[l]=a[j];
        a[j]=v;

        return(j);

}
```

Sample Output:

Best Case:

```
Enter the number of elements: 12

Enter array elements:10 20 30 40 50 60 70 80 90 100 110 120

New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
Total no. of comparisons: 88
Total no. of swaps: 11
```

Worst case:

```
Enter the number of elements: 12

Enter array elements:120 110 100 90 80 70 60 50 40 30 20 10

New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
Total no. of comparisons: 88
Total no. of swaps: 11
```

Average Case:

```
Enter the number of elements: 12

Enter array elements:10 70 20 80 30 90 40 100 50 110 60 120

New Phase:
10 20 30 40 60 50 70 100 90 110 80 120
New Phase:
10 20 30 40 50 60 70 100 90 110 80 120
New Phase:
10 20 30 40 50 60 70 100 90 110 80 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
New Phase:
10 20 30 40 50 60 70 80 90 100 110 120
Total no. of comparisons: 49
Total no. of swaps: 12
```

Conclusion: Thus, from this experiment we learn Quick sort it turns out to be the fastest sorting algorithm in practice. It has a time complexity of $\Theta(n \log(n))$ on the average. However, in the (very rare) worst case quicksort is as slow as Bubble sort, namely in $\Theta(n^2)$.