Lecture 4: Mergesort and Quicksort

30th August, 2023

Merge Sort

Dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

⇒ Merge Sort solves the problem using **Divide and Conquer** approach.
⇒ A typical Divide and Conquer algorithm solves a problem using following three steps:

1. **Divide:** This involves dividing the problem into smaller sub-problems.
2. **Conquer:** Solve sub-problems by calling recursively until solved.
3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.

Pseudo Code

```
function sort(array){
    if len(array) == 0:
        return;

    sort(array[0   : n/2]);
    sort(array[n/2 : n]);

    merge(array, 0, n/2, n);
}
```

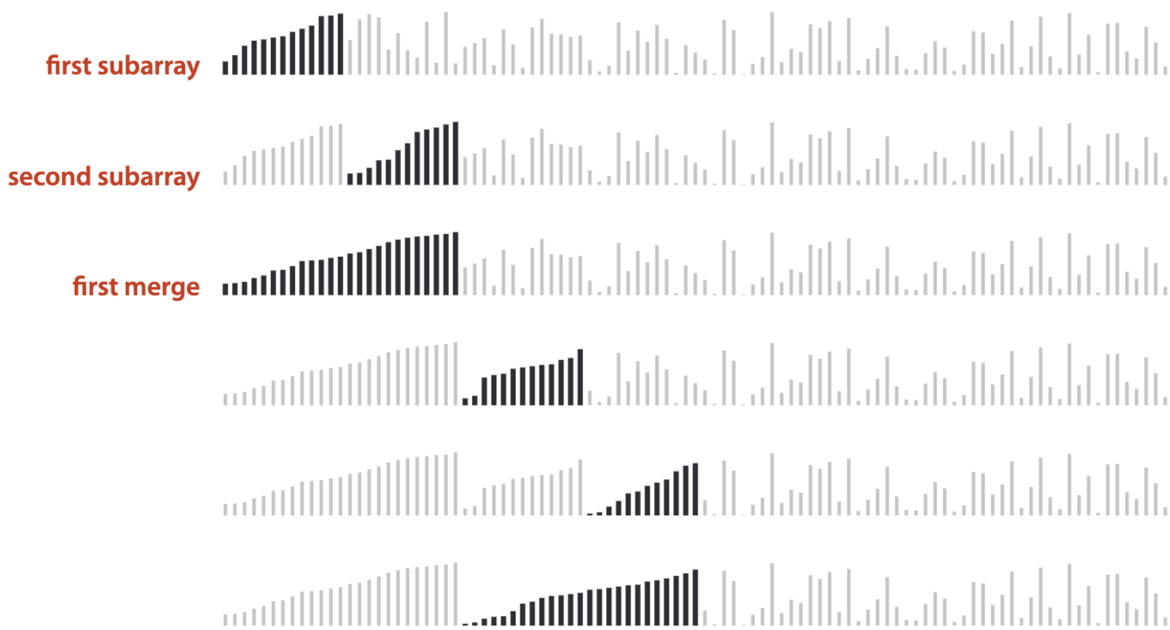
Implementation

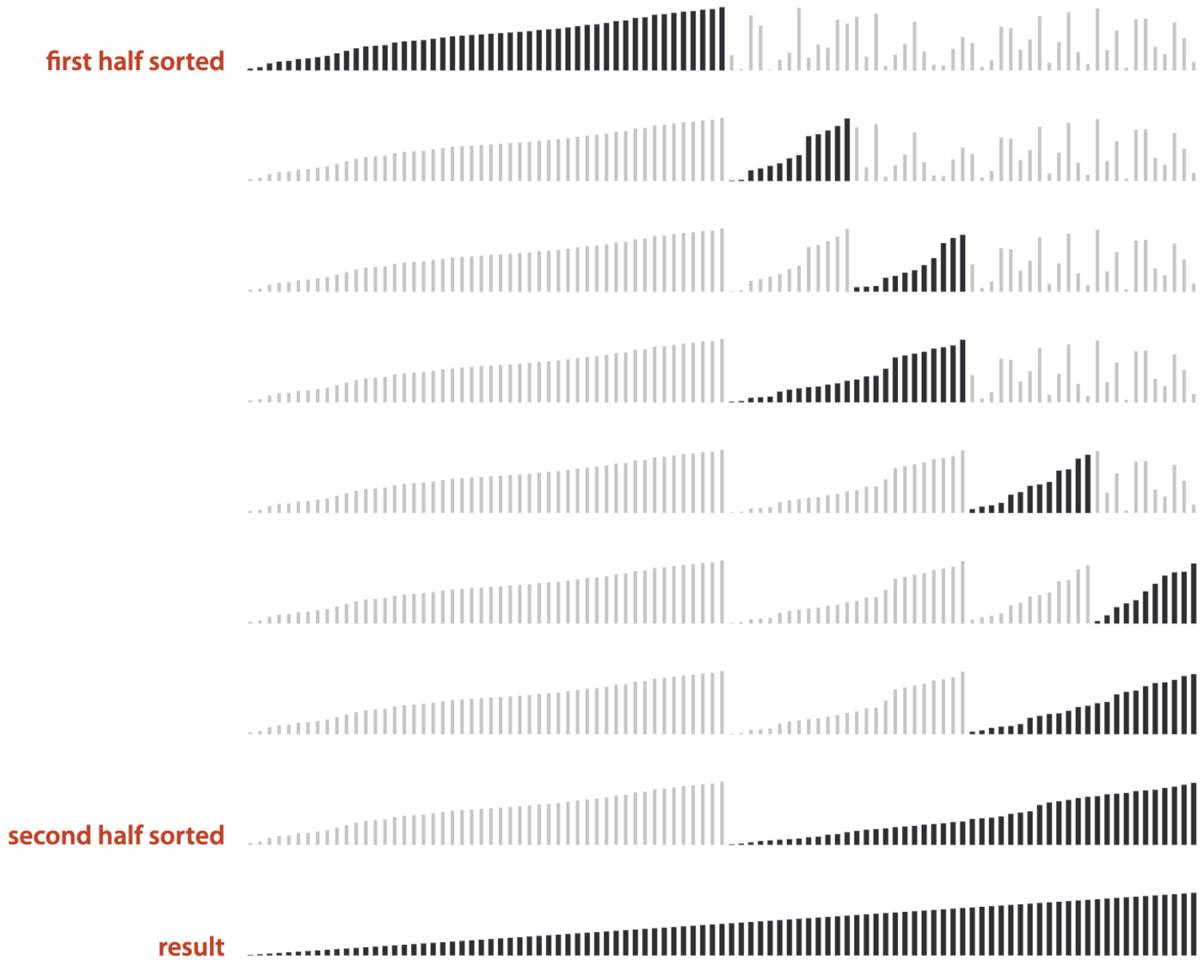
```
void merge(vector<int>& nums, int low, int mid, int high){
   // copying nums[low:high] => aux[low:high]
   for(int i=low; i<=high; i++){
       aux[i] = nums[i];
   }

   // this variables will refer to where we are in sub-array
   // initially, both point to the first element in respective sub-arrays
   int left_ind = low;
   int right_ind = mid+1;
   int curr_ind = low;

   // iterate through all position and pick the correct element
   for(int ptr=low; ptr<=high; ptr++){
       // if left subarray exhausted then use right one
       if(left_ind > mid){
           nums[ptr] = aux[right_ind];
           right_ind += 1;
       }
       // if right subarray exhausted then use left one
       else if(right_ind > high){
           nums[ptr] = aux[left_ind];
           left_ind += 1;
       }
       // if we have elements on both subarrays
       //choose the left if its smaller
       else if(aux[left_ind] < aux[right_ind]){
           nums[ptr] = aux[left_ind];
           left_ind += 1;
       }
       // choose right if its smaller
       else{
           nums[ptr] = aux[right_ind];
           right_ind += 1;
       }
   }
}
```

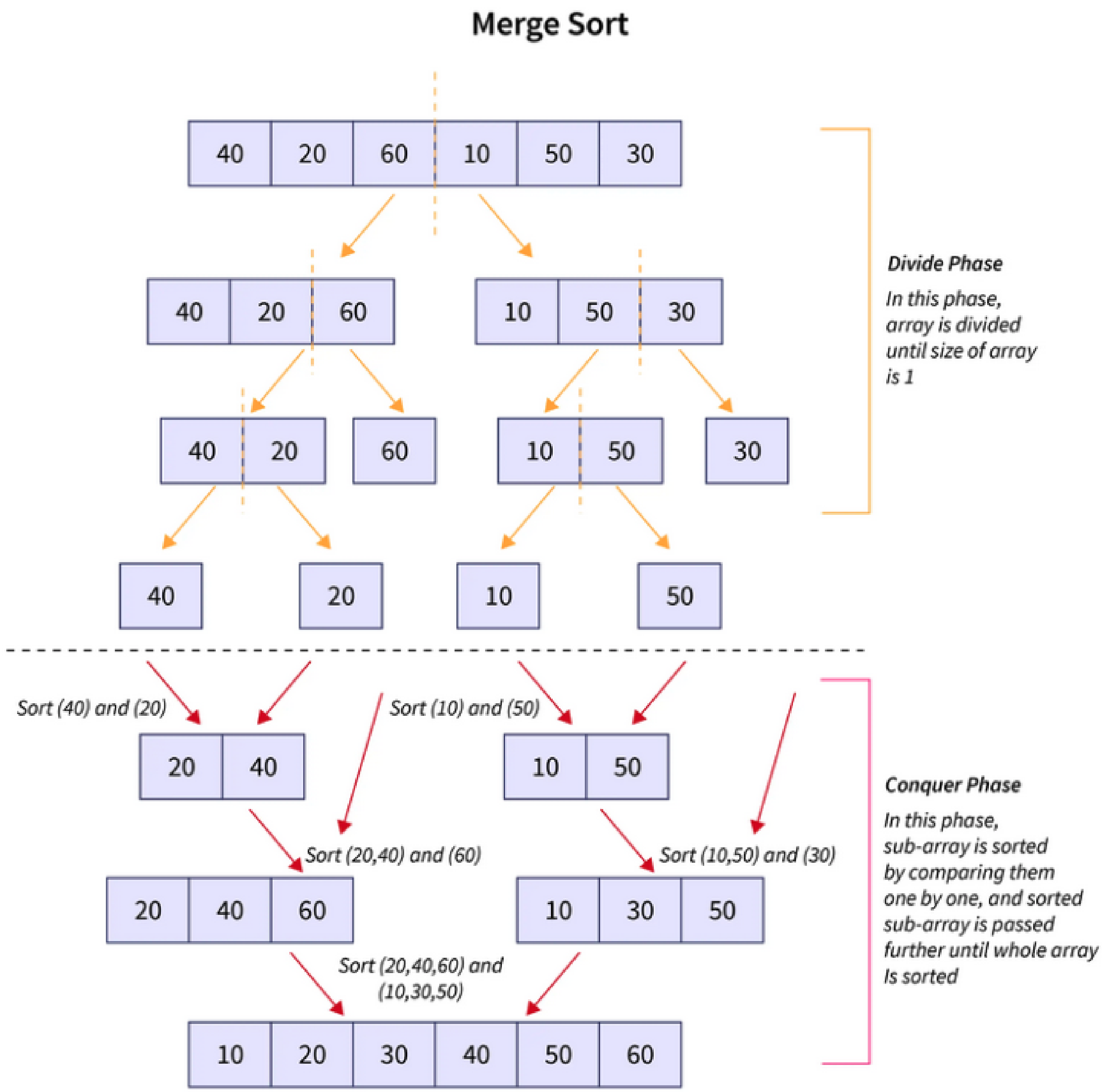
```
void merge_sort(vector<int>& nums, int low, int high){
   // check if the bounds are correct
```

```
    if (low >= high) return;

    // finding the split index
    int mid = (low + high)/2;

    // recursively sorting two sub-arrays
    merge_sort(nums, low,   mid);
    merge_sort(nums, mid+1, high);

    // merge the two sorted arrays
    merge(nums, low, mid, high);
}
```

Example and Visualization




first half sorted
second half sorted
result



Complexity Analysis

Time Complexity

We can not do simple analysis as we have been doing in previous algorithms. As there are no for loops or while loops in the implementation.
But there is **recursion**.

⇒ recursion here is running the merge algorithm multiple times. (We don’t know how many times)

⇒ We can write,

T(sort array of size n) = 2*T(sort array of size n/2) + T(merge)

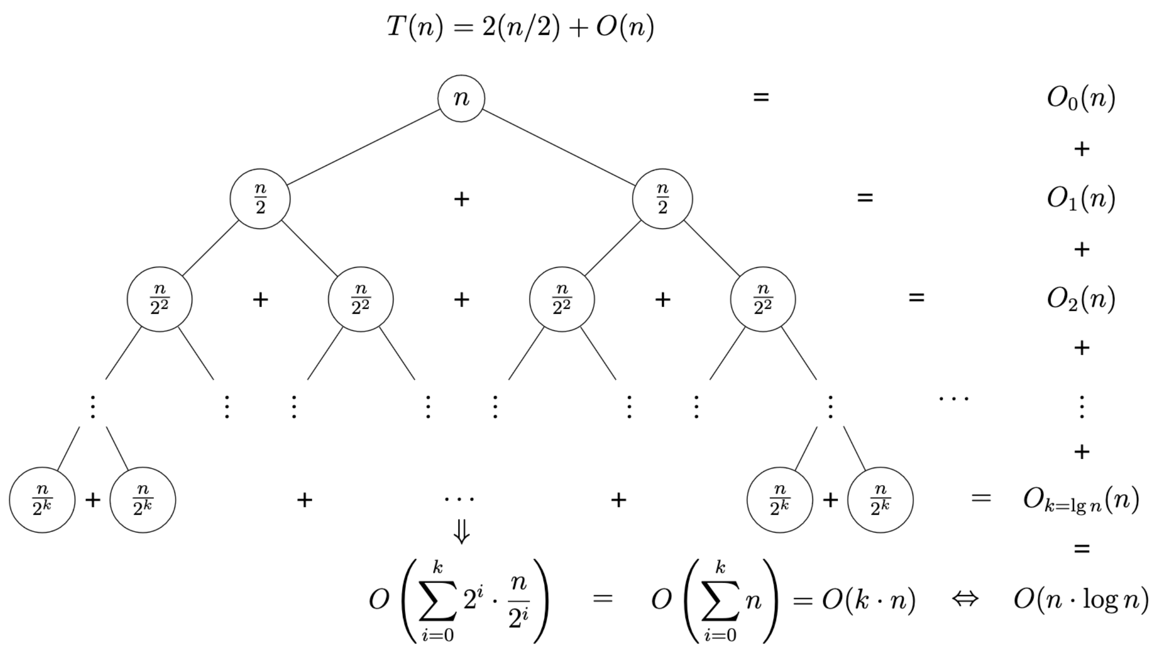
⇒ Let’s first find the time complexity of merge algorithm.
⇒ Analyzing the merge algorithm is not that hard. We can see that there are not any cases where the running time of for loop might change. So, we can say that the merge algorithm will always take, **n steps**

⇒ So now,
T(sort array of size n) = 2*T(sort array of size n/2) + n

⇒ To find how much time the recursion will take, we have multiple methods.

- Substitution Method
- Recursion Tree
- Master Method

Let’s look at how recurrence tree method works,



Space Complexity

The merge sort algorithm is not actually using any extra space, but the merge algorithm requires an auxiliary array to merge two sorted arrays.

- Input Time Complexity: θ(n)
- Output Time Complexity: θ(1)
- Auxiliary Time Complexity: θ(n)

Quicksort

It works by partitioning an array into two subarrays, then sorting the subarrays independently.
⇒ It is a Divide and Conquer approach.
⇒ Quicksort is popular because it is not difficult to implement, works well for a variety of different kinds of input data, and is substantially faster than any other sorting method in typical applications.

Pseudo Code

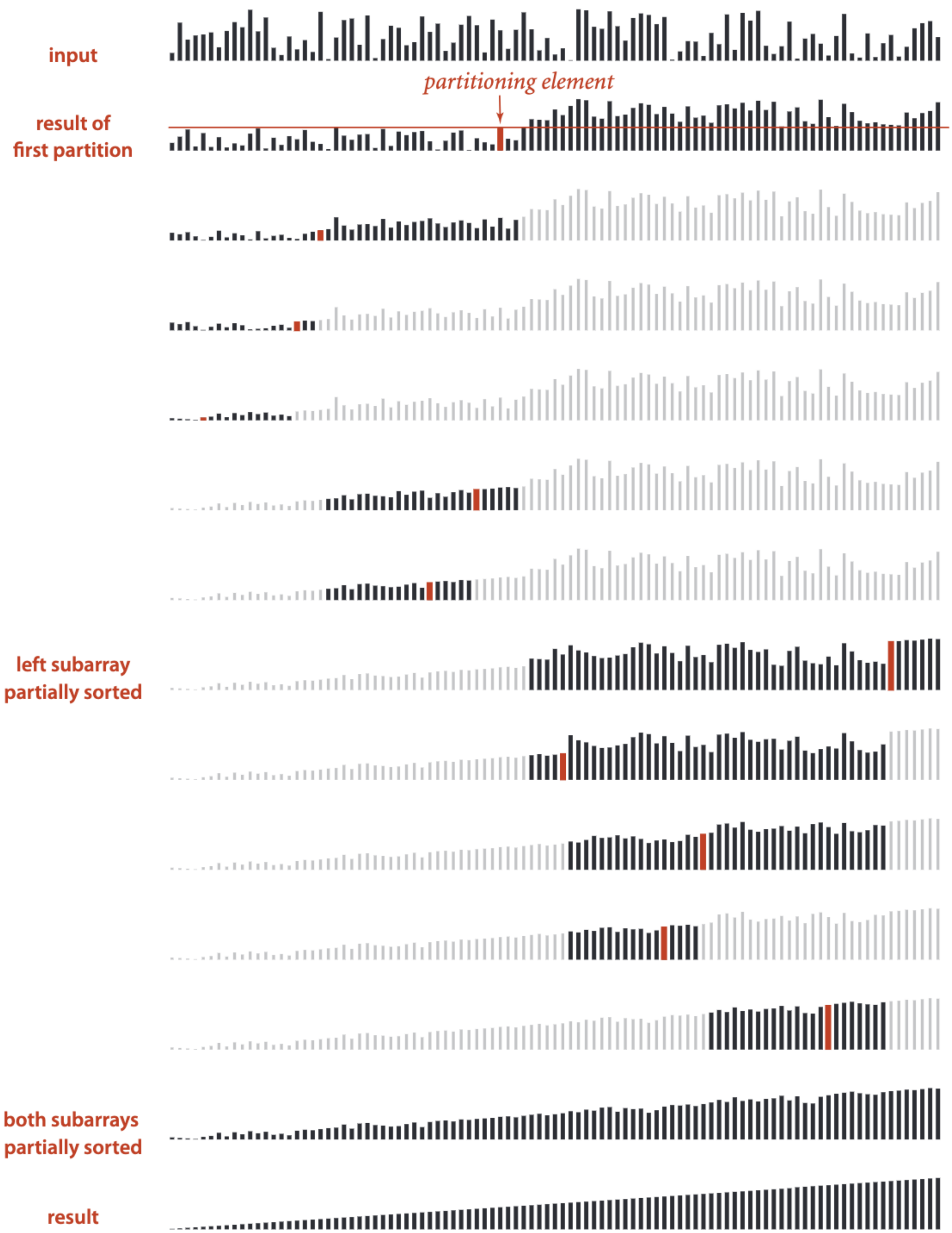
```
function quick_sort(array){
      pivot_index = partition(array);

      quick_sort(array[0             : pivot_index])
      quick_sort(array[pivot_index+1 : n])
}
```

Implementation

```cpp
int partition(vector<int>& nums, int low, int high){
   int pivot = low;
   int low_index = pivot + 1;

   // iterating through all elements and putting them in right set
   for(int i=pivot+1; i<=high; i++){
       if( nums[i] < nums[pivot] ){
           swap(nums[i], nums[low_index]);
           low_index+=1;
       }
   }

   // move the pivot in correct position
   swap(nums[pivot], nums[low_index-1]);

   return low_index-1;
}
```

```cpp
void quick_sort(vector<int>& nums, int low, int high){
   if( low >= high ) return;

   int partition_index = partition(nums, low, high);

   quick_sort(nums, low,               partition_index-1);
   quick_sort(nums, partition_index+1, high);
}
```

Example and Visualization


input
partitioning element
result of first partition
left subarray partially sorted
both subarrays partially sorted
result

Complexity Analysis

Time Complexity

We can write expression of steps taken by quick sort as below,
T(sort array of size n) = T(sort array of size x) + T(sort array of size y) + T(partition array of size n)

⇒ Before solving the recursion, let's find out the time taken by the partition of array.
⇒ There is only one for loop which runs for θ(high-low) times, here high can be n-1 and low can be 0 initially, so θ(n-1-0) = θ(n).

T(sort array of size n) = T(sort array of size x) + T(sort array of size y) + n

⇒ Here, we can see that the problem is not divided in the same size (as in the merge sort)
⇒ So, let's do worst-case and best-case analysis

Worst Case

If the input was, nums = [5, 4, 3, 2, 1]
The algorithm will take 5 as the pivot element and partition it, which will look something like this (after partition),

nums = [4, 3, 2, 1, **5**]

So, left of pivot element (here 5), all elements which are less than 5. and right of the pivot element is all elements which are bigger than 5.
The size of these two sides are **not** similar. (there are 4 elements on the left side and 0 elements on the right side).

Now, when running quick-sort for a smaller array, [4, 3, 2, 1], pivot element will be 4. So, after partition,

nums = [3, 2, 1, **4**]

Again, the left side has 3 elements and the right side is empty.

This pattern will happen for every iteration in that array.

⇒ Let's try to put this in expression,

1.) T(sort array of size n) = **T(sort array of size n-1) + T(sort array of size 1)** + n

now, let's substitute n with n-1

2.) T(sort array of size n-1) = **T(sort array of size n-1-1) + T(sort array of size 1)** + n-1

Put 2. in 1.

$$T(n) = (T(n-2) + T(1) + n-1) + T(1) + n$$

Now, if we continue to substitute n-2 and so on,

$$T(n) = 1 + 2 + 3 + . . . + n-1 + n$$
$$= n^2$$

⇒ **Worst Case Time Complexity: $\theta(n^2)$**

Best Case

Now, the intuition behind finding the best case is as follows,
⇒ If you noticed, our time complexity changes depending on how the algorithm splits the bigger problem into smaller ones.
⇒ If we split into <u>uneven</u> parts, then the bigger part will dominate. So, if the size of both splits are the same then none of the splits can dominate.
⇒ So, the best-case will happen when the partition always splits the array into two even parts.

$$T(n) = T(n/2) + T(n/2) + n$$

Which looks like the merge sort algorithm's time complexity.
So, using the same method as above,

$$T(n) = \theta(n*\log(n))$$

⇒ **Best Case Time Complexity: $\theta(n*\log(n))$**

Space Complexity

⇒ The quick-sort algorithm does not require any extra variables.

- Input Time Complexity: $\theta(n)$
- Output Time Complexity: $\theta(1)$
- Auxiliary Time Complexity: $\theta(1)$

Resources

- **Readings**: Algorithms, 4e. Section 2.2, 2.3
- Polls
- Merge Sort Visualization
- Quick Sort Visualization