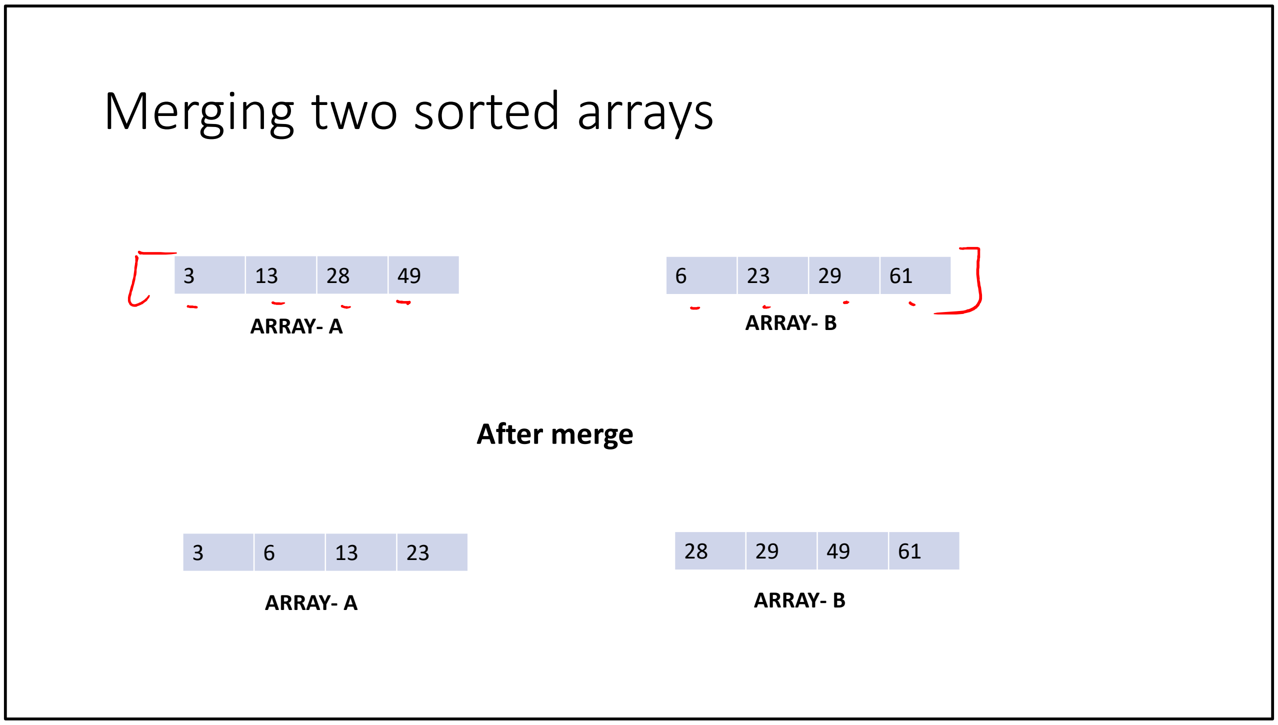
Today's class

- Merge sort
- Quick sort
- Dynamic arrays

Merge sort

References

- Read section-2.3.1 from CLRS



Merging two sorted arrays

3	13	28	49

ARRAY- A

6	23	29	61

ARRAY- B

After merge

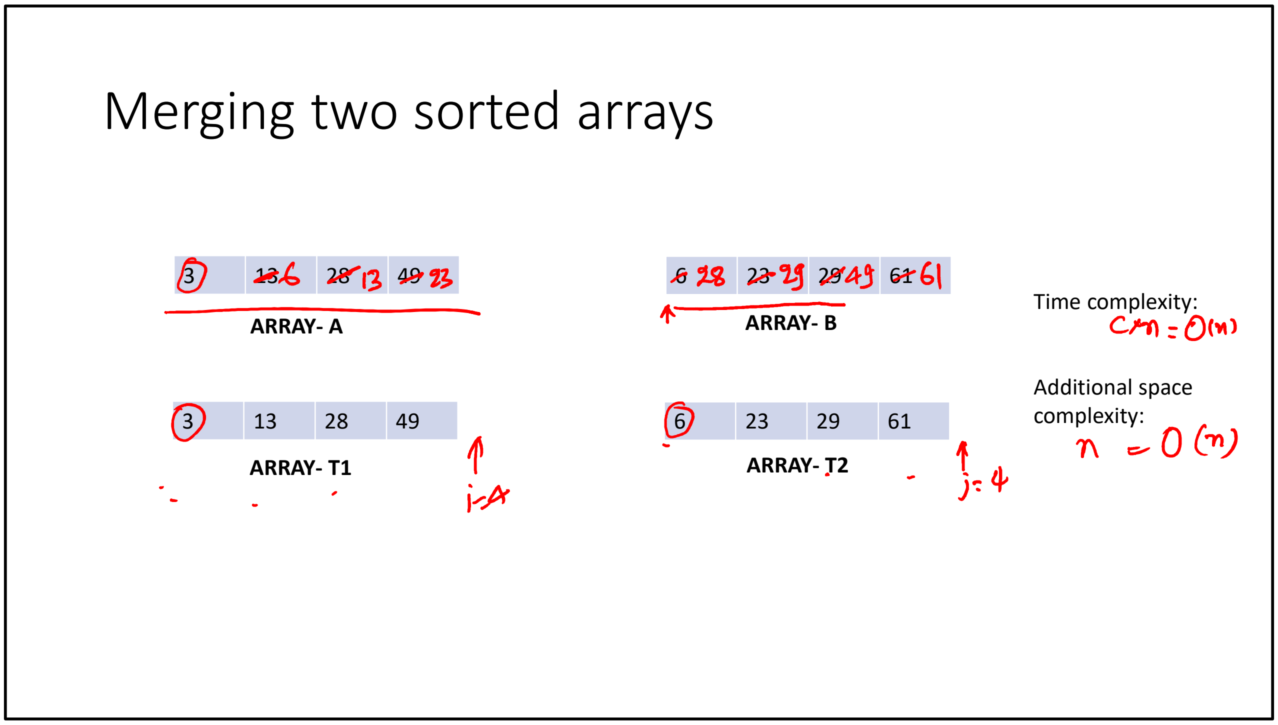
3	6	13	23

ARRAY- A

28	29	49	61

ARRAY- B

The goal here is to merge two sorted arrays, A and B, in such a way that the combined array (considering arrays A and B as a whole) is sorted.



Merging two sorted arrays

3	~~13~~ 6	~~28~~ 13	~~49~~ 23

ARRAY- A

~~6~~ 28	~~23~~ 29	~~29~~ 49	~~61~~ 61

ARRAY- B

3	13	28	49

ARRAY- T1

i=4

6	23	29	61

ARRAY- T2

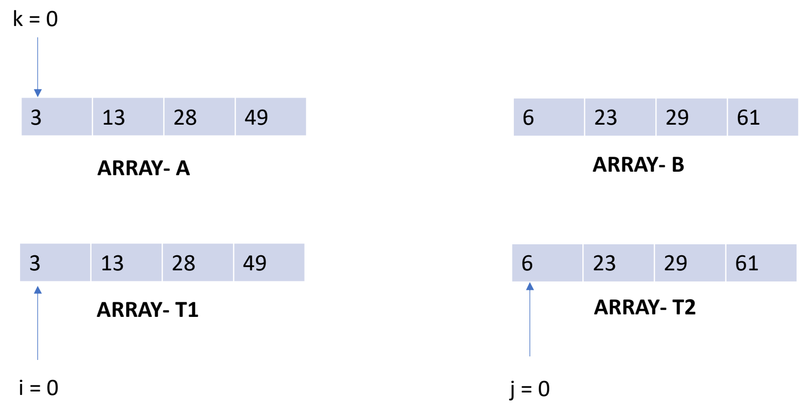
j=4

Time complexity:
$c*n = O(n)$

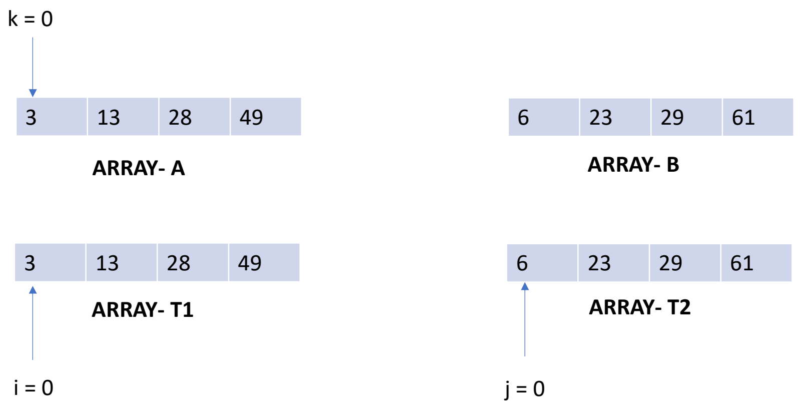
Additional space complexity:
$n = O(n)$

This algorithm creates copies of arrays A and B, compares the values in the temporary arrays, copies the smaller value into the original array, and skips the copied value in the temporary array. If the sum of the number of elements in arrays T1 and T2 is n, then this procedure makes n comparisons in the worst case. In addition, it performs 2 * n copy operations (copying from the original to temporary and vice versa). The time complexity of this algorithm is O(n). The following slides demonstrate the algorithm in detail.

Merging two sorted arrays

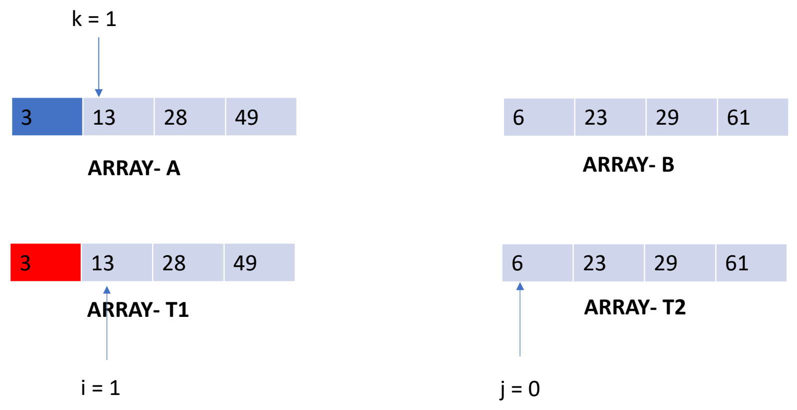


Merging two sorted arrays



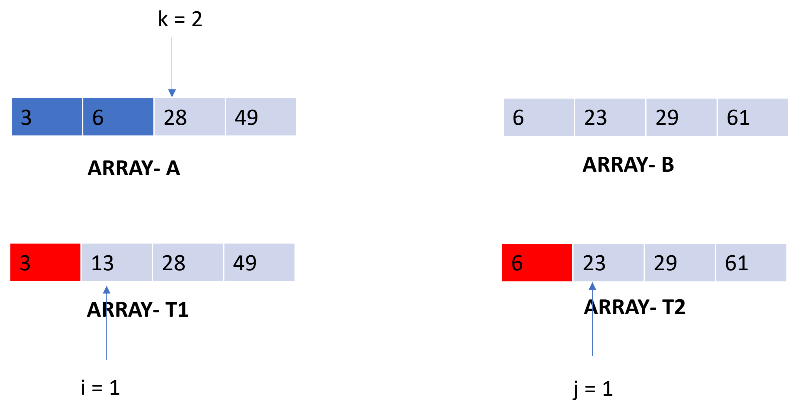
```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

Merging two sorted arrays

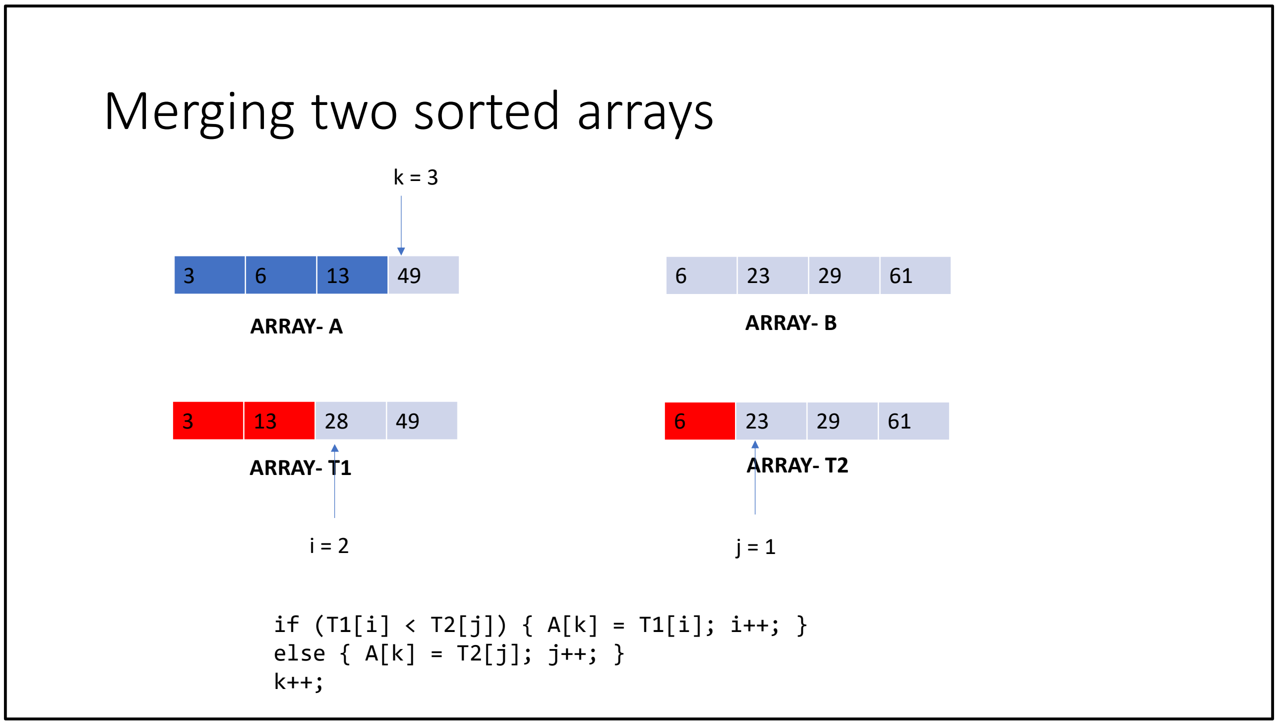


```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

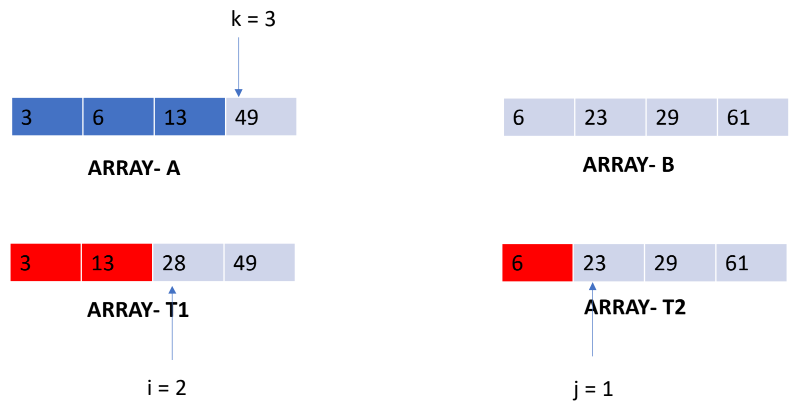
Merging two sorted arrays



```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

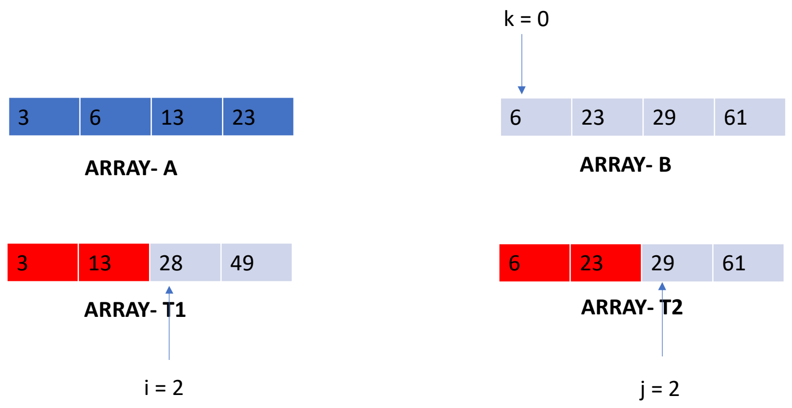


Merging two sorted arrays



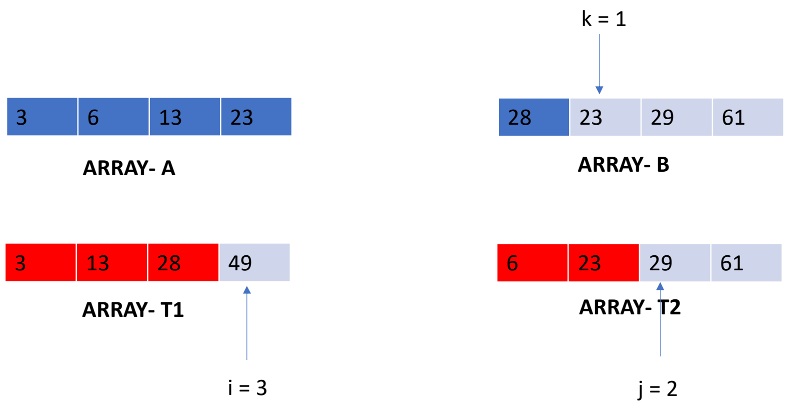
```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

Merging two sorted arrays



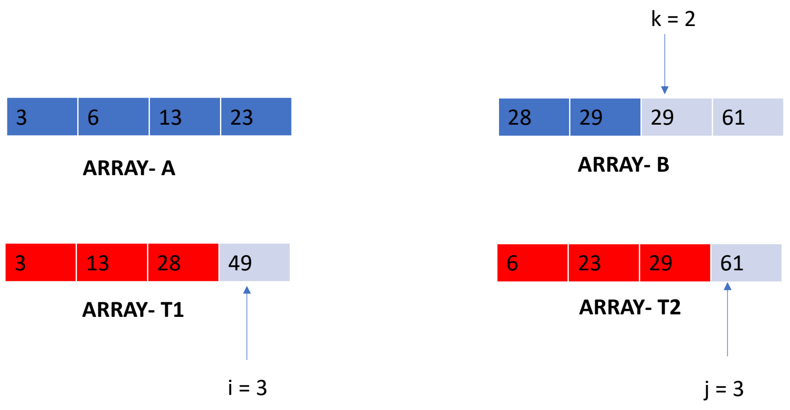
```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

Merging two sorted arrays



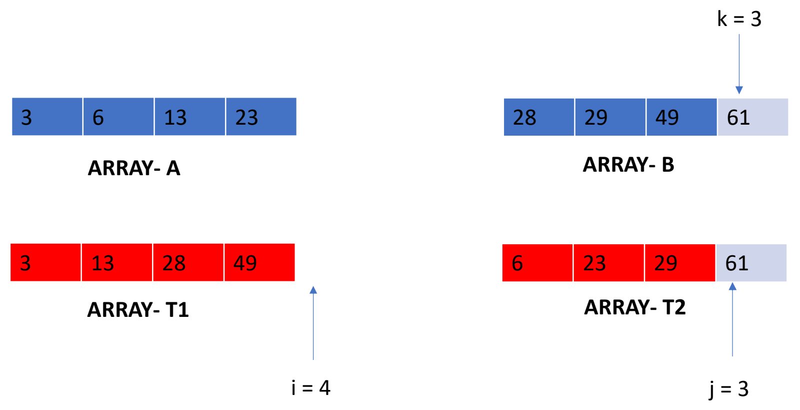
```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

Merging two sorted arrays



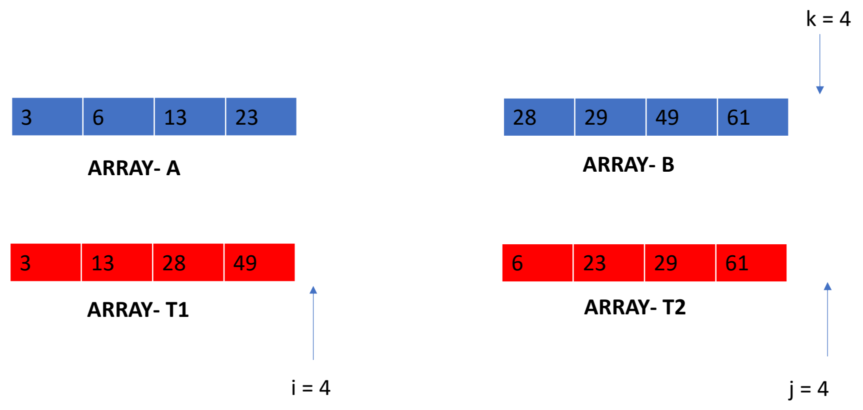
```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

Merging two sorted arrays



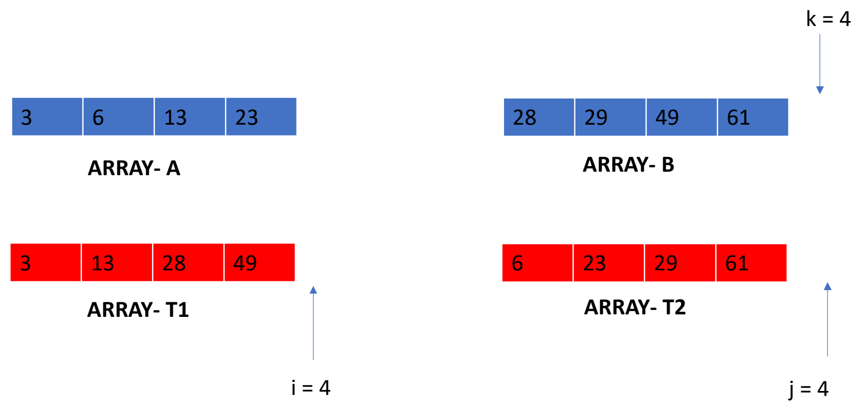
```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

Merging two sorted arrays



```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

Merging two sorted arrays



Time complexity:
c * n

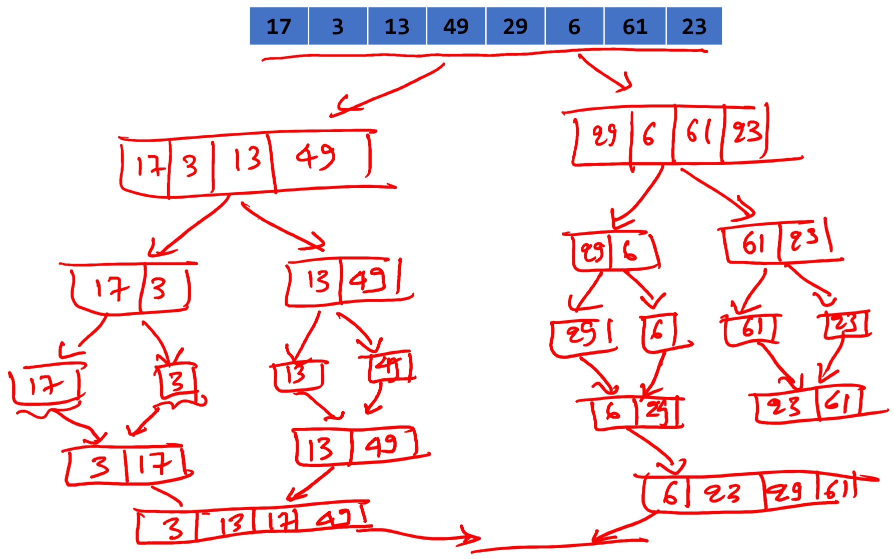
Additional space complexity:
n

```
if (T1[i] < T2[j]) { A[k] = T1[i]; i++; }
else { A[k] = T2[j]; j++; }
k++;
```

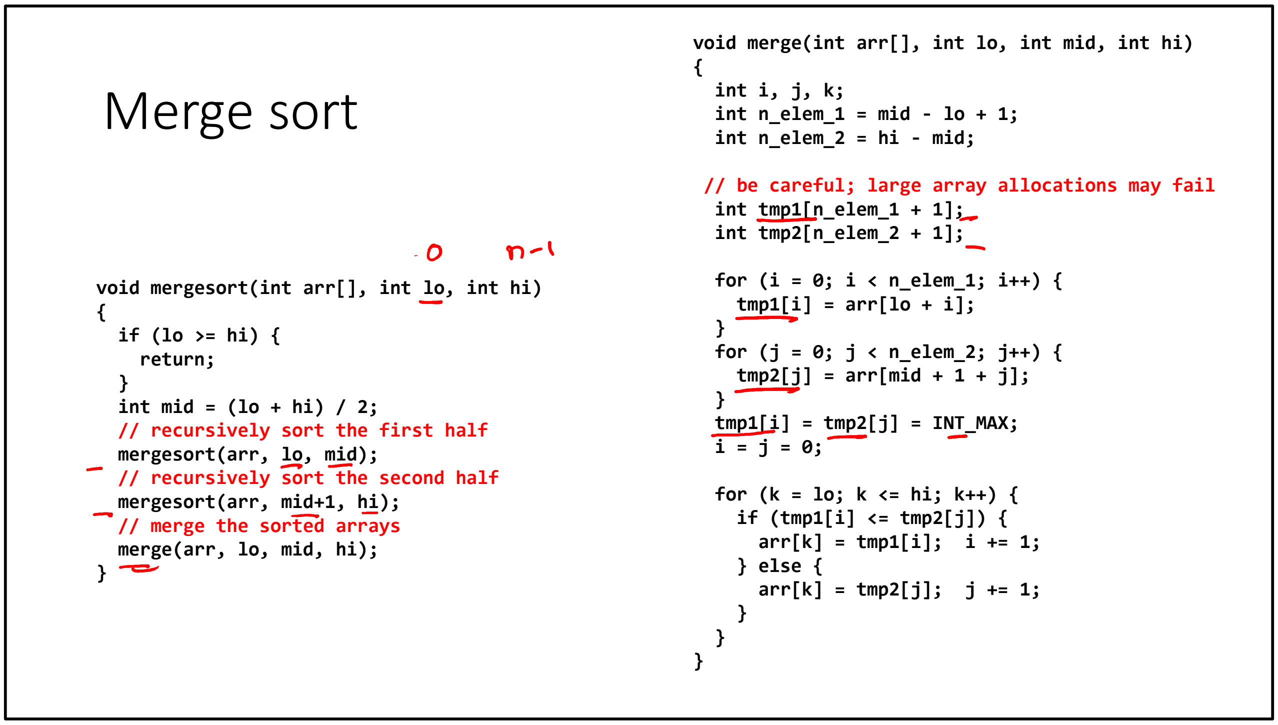
Merge sort

- Merge sort is a recursive algorithm
- Base condition:
  - if the input array has one or zero elements, the array is already sorted; return the array
- Recursive step
  - Divide the array into two sub-arrays, A_1 and A_2
    - A_1 contains the first half, A_2 contains the second half
  - Recursively sort A_1 and A_2
  - Merge sorted A_1 and A_2 into a single sorted array

Merge sort



17 3 13 49 29 6 61 23
17 3 13 49
29 6 61 23
17 3
13 49
29 6
61 23
17
3
13
49
29
6
61
23
3 17
13 49
6 29
23 61
3 13 17 49
6 23 29 61



Merge sort

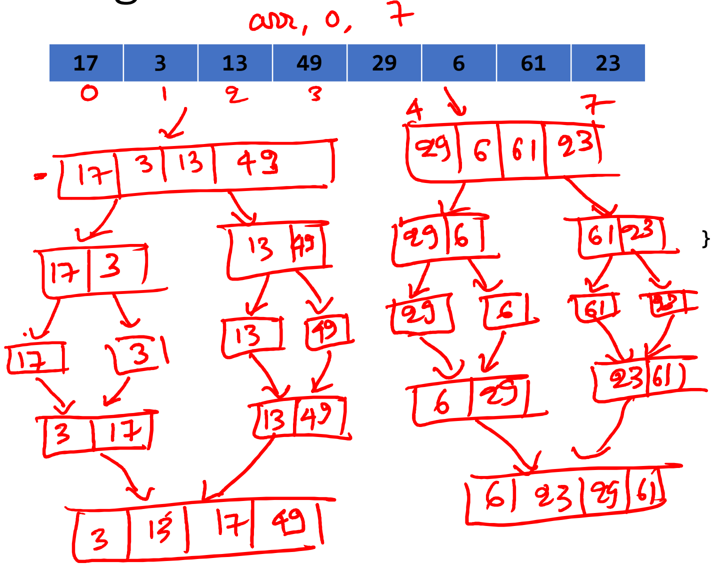
0 n-1

```
void mergesort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int mid = (lo + hi) / 2;
  // recursively sort the first half
  mergesort(arr, lo, mid);
  // recursively sort the second half
  mergesort(arr, mid+1, hi);
  // merge the sorted arrays
  merge(arr, lo, mid, hi);
}
```

```
void merge(int arr[], int lo, int mid, int hi)
{
  int i, j, k;
  int n_elem_1 = mid - lo + 1;
  int n_elem_2 = hi - mid;

 // be careful; large array allocations may fail
  int tmp1[n_elem_1 + 1];
  int tmp2[n_elem_2 + 1];

  for (i = 0; i < n_elem_1; i++) {
    tmp1[i] = arr[lo + i];
  }
  for (j = 0; j < n_elem_2; j++) {
    tmp2[j] = arr[mid + 1 + j];
  }
  tmp1[i] = tmp2[j] = INT_MAX;
  i = j = 0;

  for (k = lo; k <= hi; k++) {
    if (tmp1[i] <= tmp2[j]) {
      arr[k] = tmp1[i];  i += 1;
    } else {
      arr[k] = tmp2[j];  j += 1;
    }
  }
}
```

Initially, mergesort is called with mergesort(arr, 0, n-1), where n is the number of elements in the arr. You need to be careful during the static allocation of large arrays (like the one we are doing in the merge procedure) because they may fail without any warning. Dynamic memory allocations, as we will study in the following classes, are another option, but dynamically allocating and deallocating memory in the merge routine is also a bad idea because dynamic allocations can be much more expensive than static allocations. A better strategy would be to allocate a temporary array that can hold n elements in the "main" routine or the caller of mergesort and pass it to the merge routine. The temporary array can be deleted when the mergesort finally returns to its original caller (e.g., main). The mergesort routine recursively divides the array into two parts until the parts contain only one element. At this point, the parts are already sorted (because an unsorted one-element array doesn't exist). Then before returning, the algorithm merges the sorted parts in a way that the merged array is also sorted (we have discussed a strategy to merge two sorted arrays before). This divide-and-merge approach continues until the entire array is sorted. The next slides show the sequence of events in more detail.

Merge sort



```
void mergesort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int mid = (lo + hi) / 2;
  // recursively sort the first half
  mergesort(arr, lo, mid);
  // recursively sort the second half
  mergesort(arr, mid+1, hi);
  // merge the sorted arrays
  merge(arr, lo, mid, hi);
}
```

4 7

T

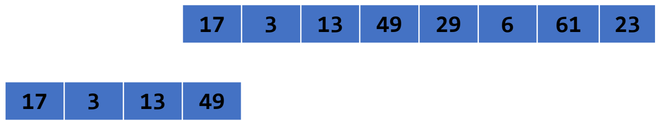
Merge sort

Divide phase: BLUE
Merge phase: RED

17	3	13	49	29	6	61	23

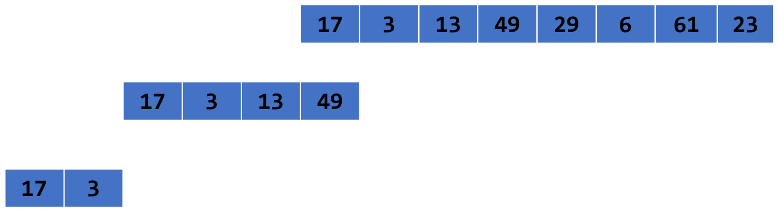
Merge sort

Divide phase: BLUE
Merge phase: RED



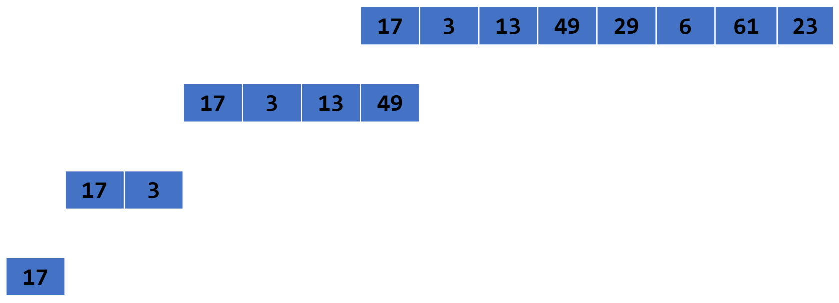
Merge sort

Divide phase: BLUE
Merge phase: RED



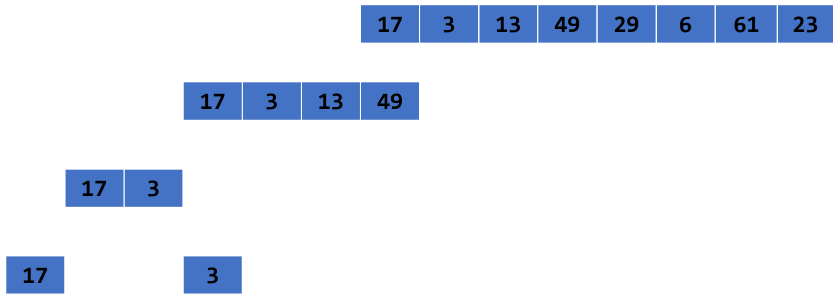
Merge sort

Divide phase: BLUE
Merge phase: RED



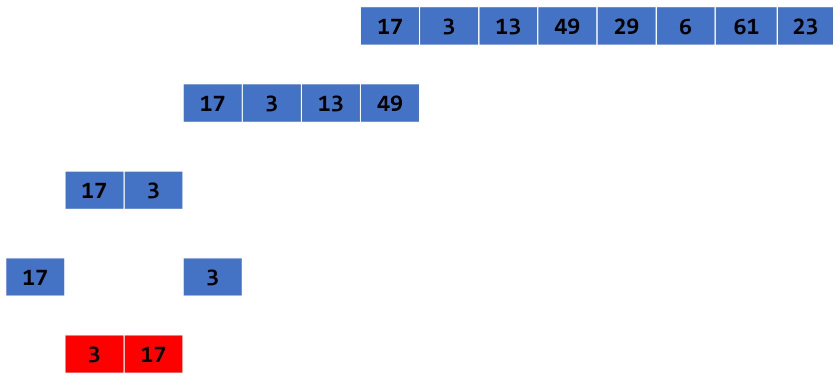
Merge sort

Divide phase: BLUE
Merge phase: RED



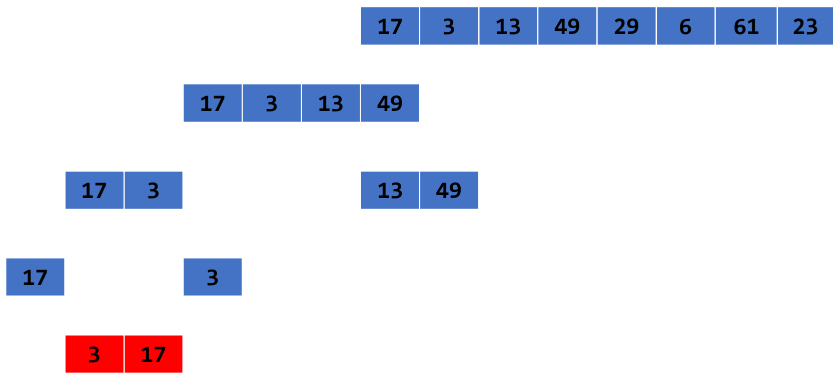
Merge sort

Divide phase: BLUE
Merge phase: RED



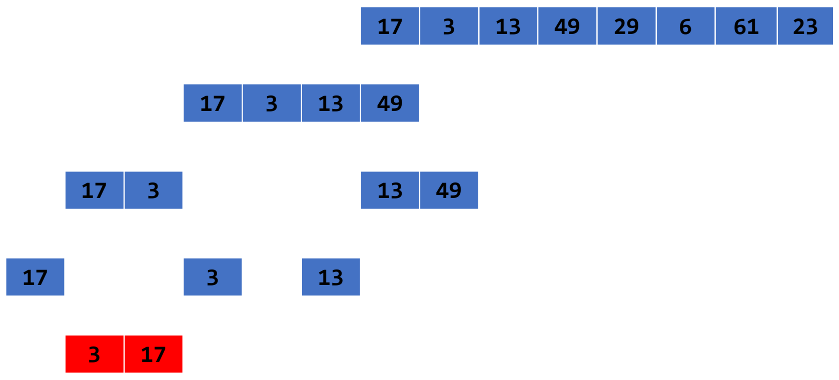
Merge sort

Divide phase: BLUE
Merge phase: RED



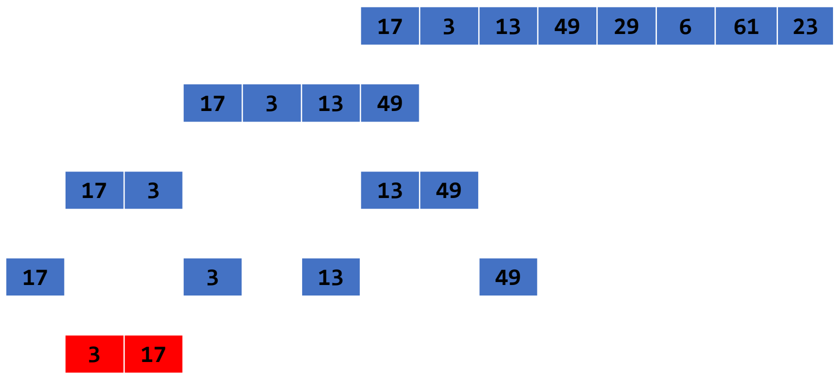
Merge sort

Divide phase: BLUE
Merge phase: RED



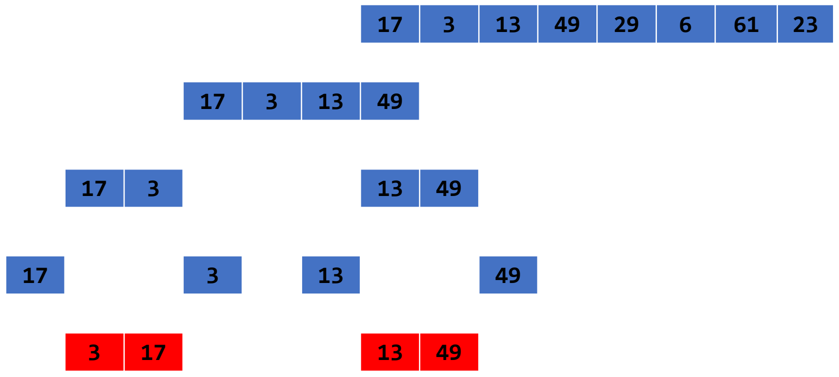
Divide phase: BLUE
Merge phase: RED

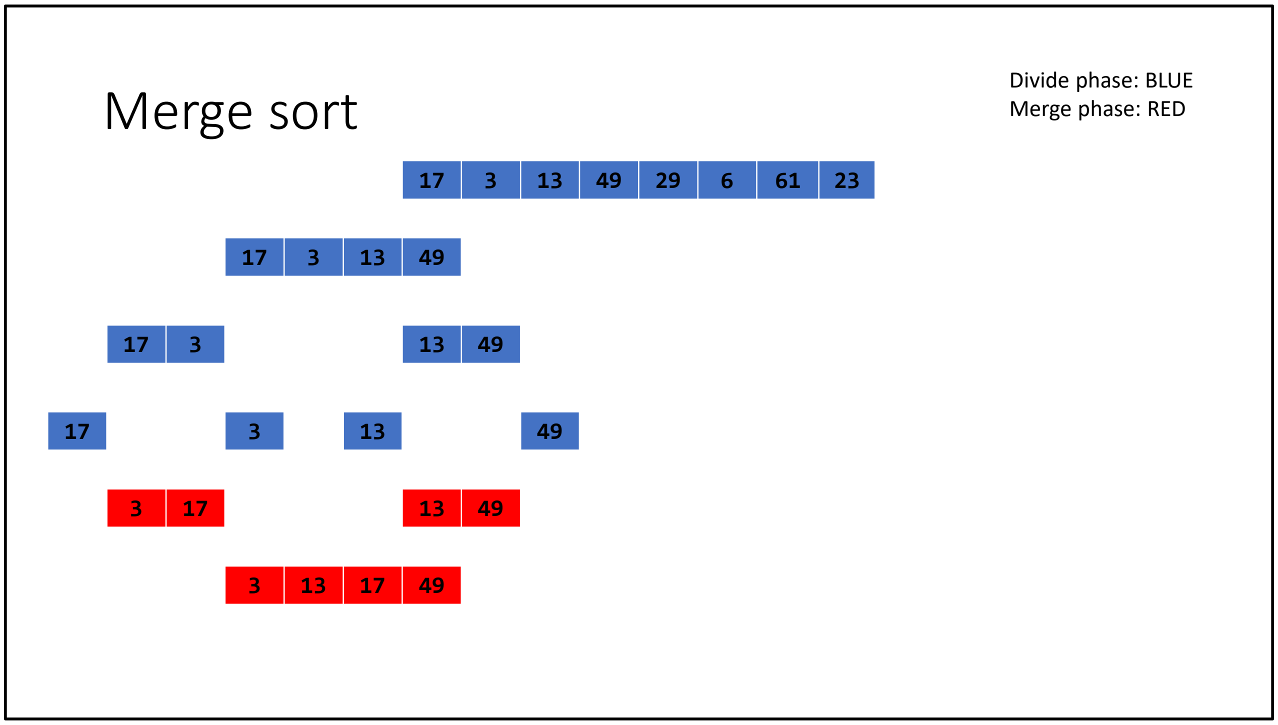
Merge sort



Merge sort

Divide phase: BLUE
Merge phase: RED



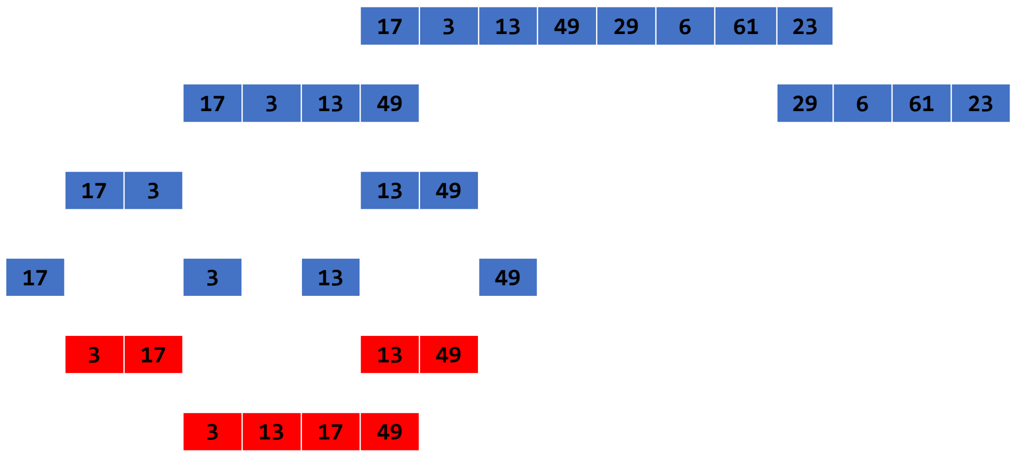


Merge sort

Divide phase: BLUE
Merge phase: RED

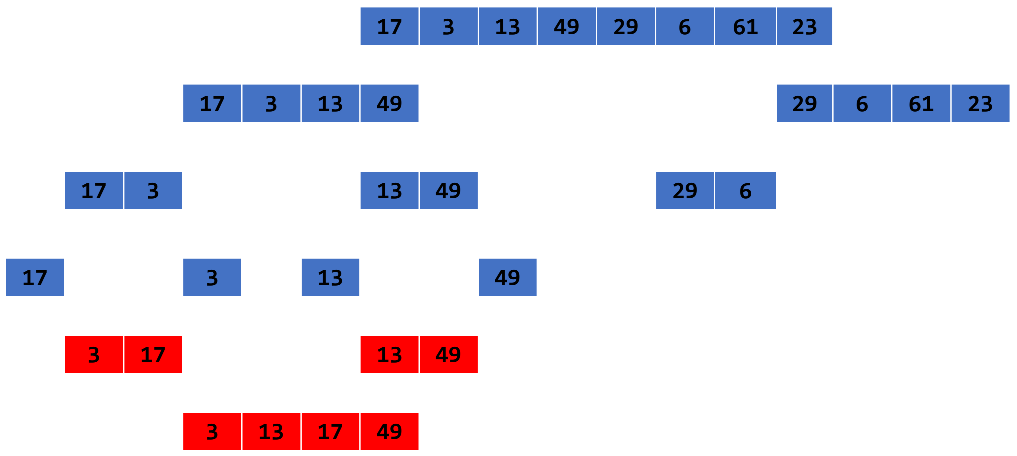
Merge sort

Divide phase: BLUE
Merge phase: RED



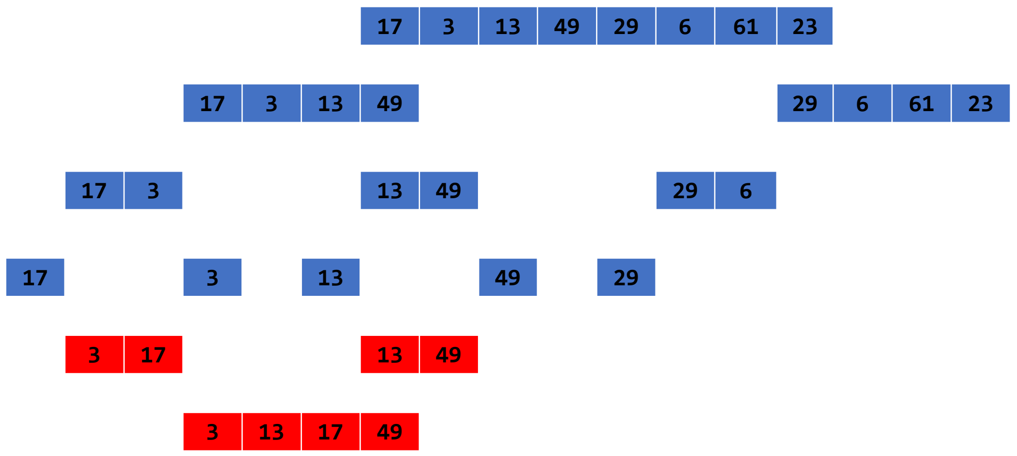
Merge sort

Divide phase: BLUE
Merge phase: RED



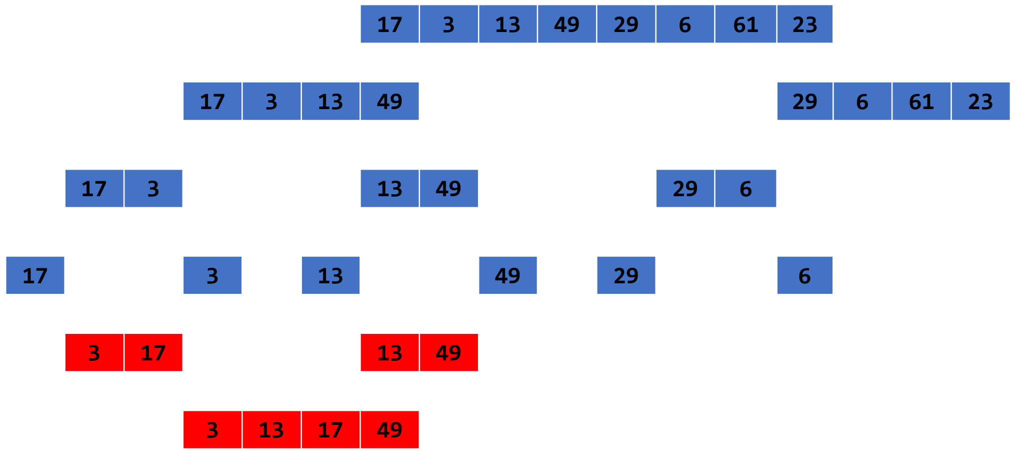
Merge sort

Divide phase: BLUE
Merge phase: RED



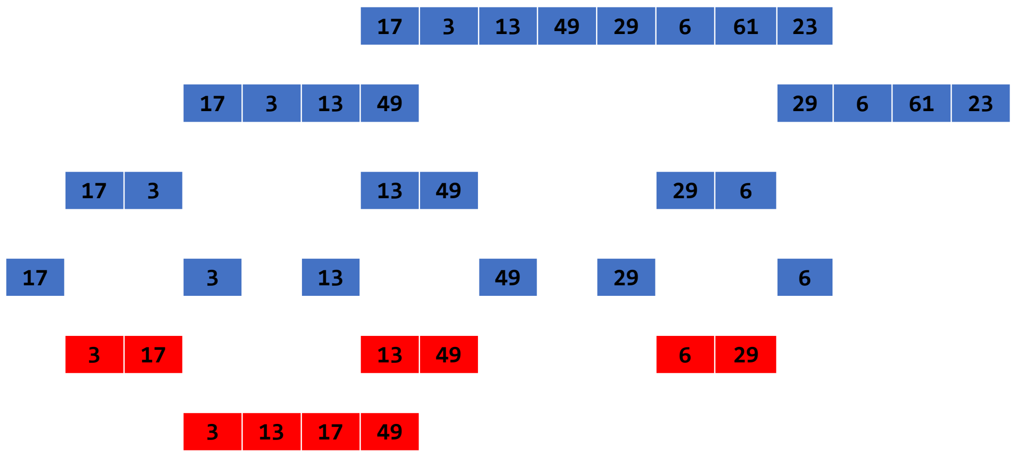
Merge sort

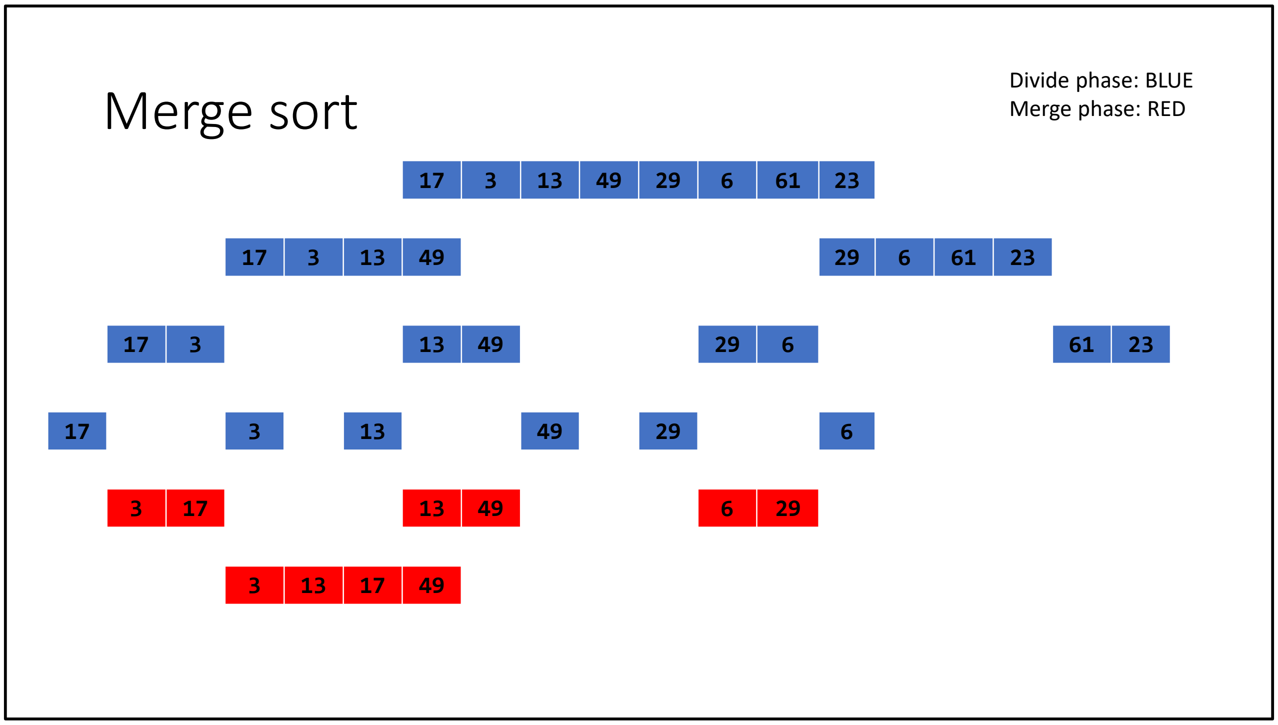
Divide phase: BLUE
Merge phase: RED



Merge sort

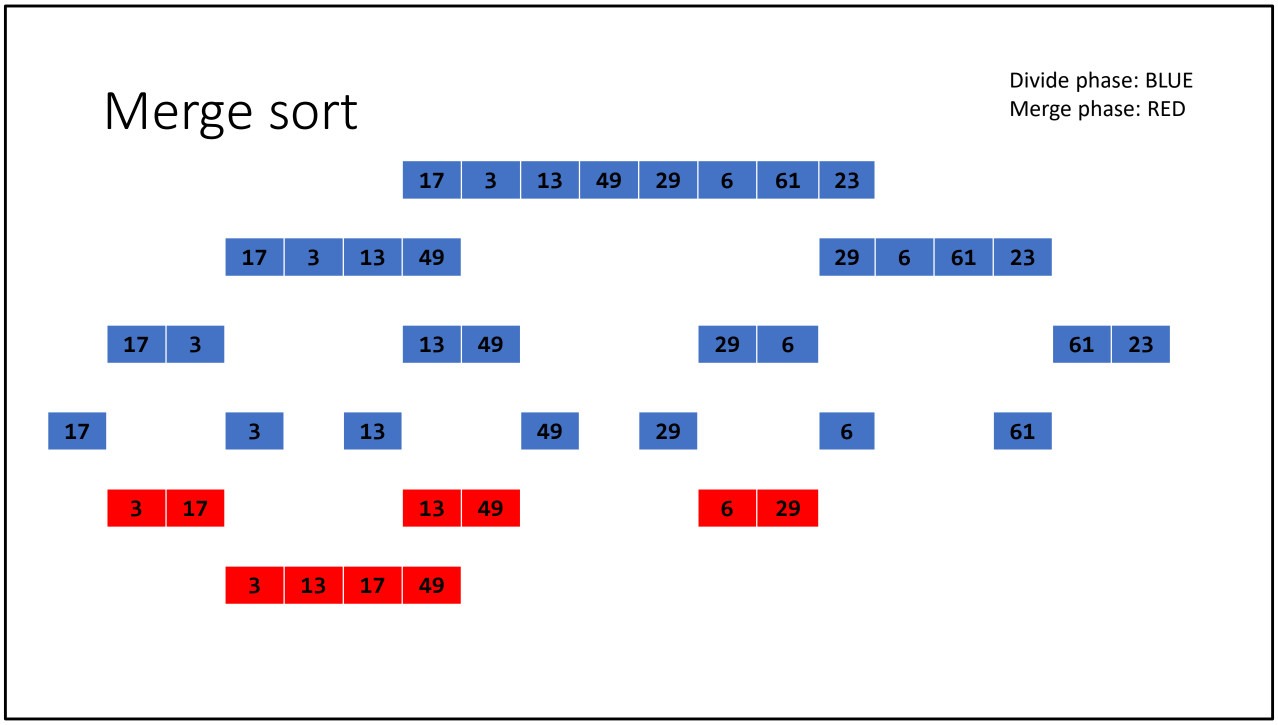
Divide phase: BLUE
Merge phase: RED





Merge sort

Divide phase: BLUE
Merge phase: RED

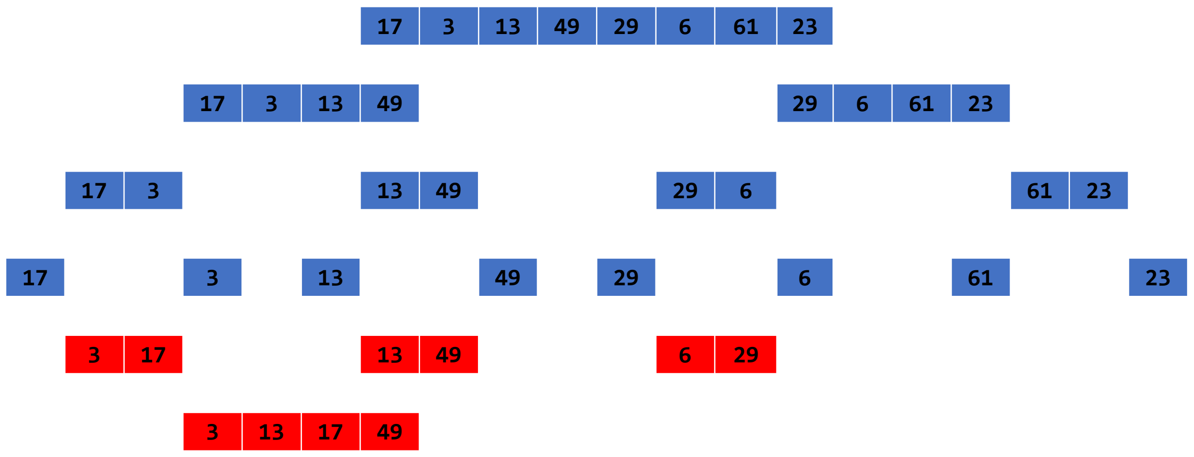


Merge sort

Divide phase: BLUE
Merge phase: RED

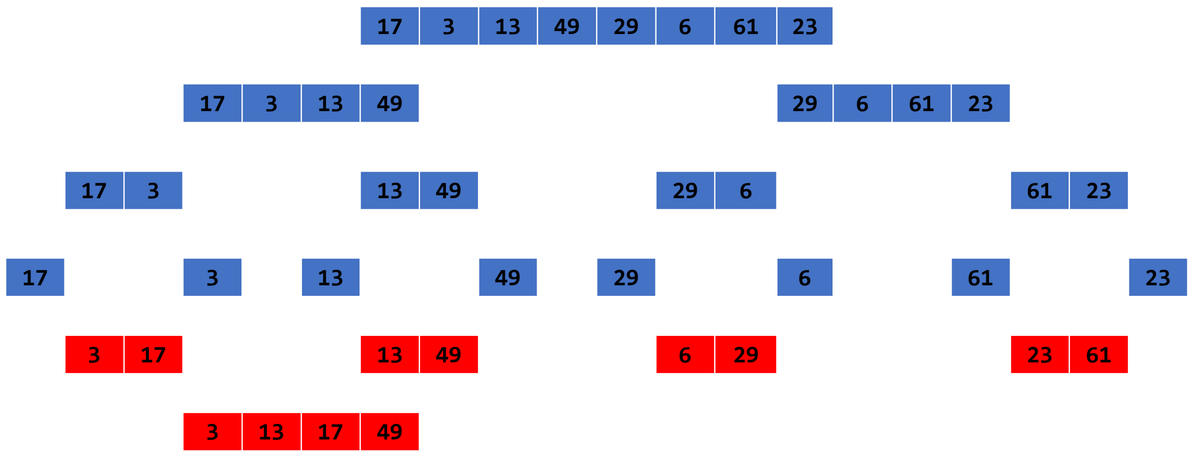
Merge sort

Divide phase: BLUE
Merge phase: RED



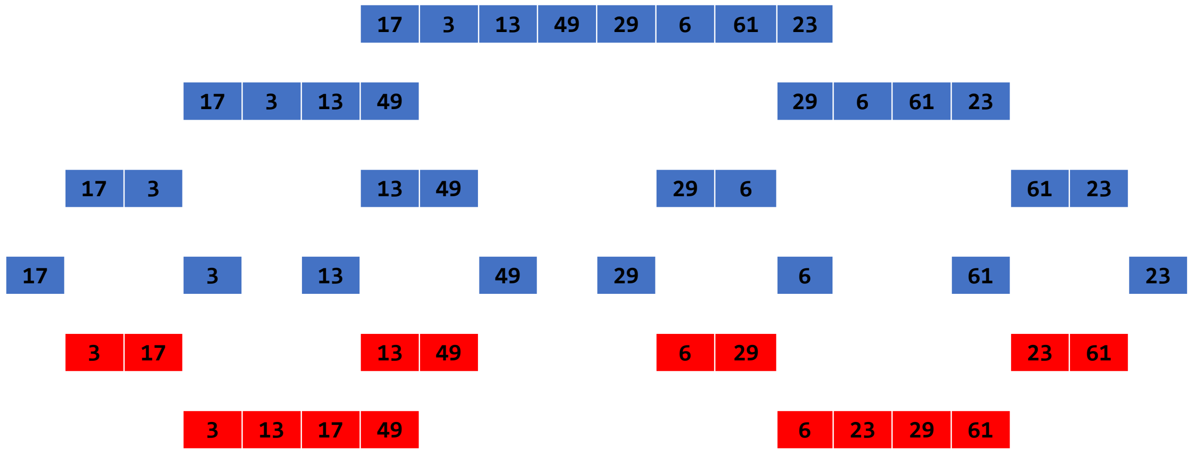
Merge sort

Divide phase: BLUE
Merge phase: RED



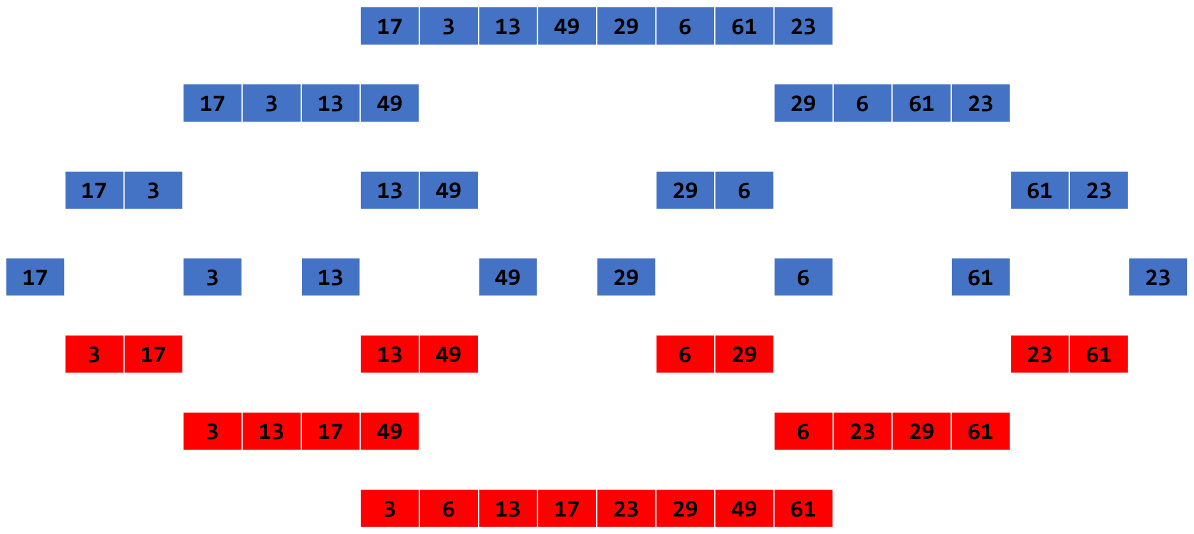
Merge sort

Divide phase: BLUE
Merge phase: RED



Merge sort

Divide phase: BLUE
Merge phase: RED



Time complexity

Merge sort

```
void mergesort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int mid = (lo + hi) / 2;
  // recursively sort the first half
  mergesort(arr, lo, mid);
  // recursively sort the second half
  mergesort(arr, mid+1, hi);
  // merge the sorted arrays
  merge(arr, lo, mid, hi);
}
```

```
void merge(int arr[], int lo, int mid, int hi)
{
  int i, j, k;
  int n_elem_1 = mid - lo + 1;
  int n_elem_2 = hi - mid;

 // be careful; large array allocations may fail
  int tmp1[n_elem_1 + 1];
  int tmp2[n_elem_2 + 1];

  for (i = 0; i < n_elem_1; i++) {
    tmp1[i] = arr[lo + i];
  }
  for (j = 0; j < n_elem_2; j++) {
    tmp2[j] = arr[mid + 1 + j];
  }
  tmp1[i] = tmp2[j] = INT_MAX;
  i = j = 0;

  for (k = lo; k <= hi; k++) {
    if (tmp1[i] <= tmp2[j]) {
      arr[k] = tmp1[i];  i += 1;
    } else {
      arr[k] = tmp2[j];  j += 1;
    }
  }
}
```

$T(1) = c$

Time complexity: $T(n) = 2T(\frac{n}{2}) + c_1 n + c_2$

The recurrence relation for time complexity is $T(n) = 2T(n/2) + c_1 * n + c_2$. This is because the two recursive calls to mergesort are done for around n/2 elements, where n is the number of elements in the array. The merge procedure does O(n) operations. Other operations in the mergesort routine are O(1).

Merge sort

- Time complexity

$T(1) = C$

Substitute. $\frac{n}{2^k} = 1$

$K = \log_2 n$

$T(n) = 2^k T(1) + K c_1 n + c_2 (2^k - 1)$

$= n * C + \log_2 n * c_1 * n + c_2 (n-1)$

$O(n \log n)$

$T(n) = 2T(n/2) + c_1 n + c_2$

$= 2\left(2T\left(\frac{n}{4}\right) + c_1 \frac{n}{2} + c_2\right) + c_1 n + c_2$

$= \underline{2^2} T\left(\frac{n}{2^2}\right) + 2c_1 n + c_2(1+2)$

$= 2^2\left(2T\left(\frac{n}{2^3}\right) + c_1 \frac{n}{2^2} + c_2\right) + 2c_1 n + c_2(1+2)$

$= \underline{2^3} T\left(\frac{n}{2^3}\right) + 3c_1 n + c_2(1+2+2^2)$

$- - - - \cdot \cdot$

kth term

$= 2^k T\left(\frac{n}{2^k}\right) + k c_1 n + c_2(1+2+2^2+ \cdots + 2^{k-1})$

Solving the recurrence relation gives us O(n * log(n)) time complexity.

Merge sort

$T(1) = 2$

$T(n) = 2T\left(\frac{n}{2}\right) + c_1 n + c_2$

$= 2\left(2T\left(\frac{n}{2^2}\right) + c_1\left(\frac{n}{2}\right) + c_2\right) + c_1 n + c_2$ // expanding T(n/2)

$= 2^2 T\left(\frac{n}{2^2}\right) + 2c_1 n + 2c_2 + c_2$ // simplifying

$= 2^2\left(2T\left(\frac{n}{2^3}\right) + c_1\left(\frac{n}{2^2}\right) + c_2\right) + 2c_2 + c_2$ // expanding T(n/2^2)

$= 2^3 T\left(\frac{n}{2^3}\right) + 3c_1 n + 2^2 c_2 + 2c_2 + c_2$ // simplifying

$= \ldots$

$= 2^k T\left(\frac{n}{2^k}\right) + kc_1 n + c_2\left(1 + 2 + 2^2 + \cdots + 2^{k-1}\right)$ // the kth term

Substitute $n = 2^k \quad \rightarrow \quad k = \log_2 n$

$T(n) = nT(1) + c_1 n log n + c_2\left(2^k - 1\right)$

$= 2n + c_1 n log n + c_2(n-1) = O(n log n)$

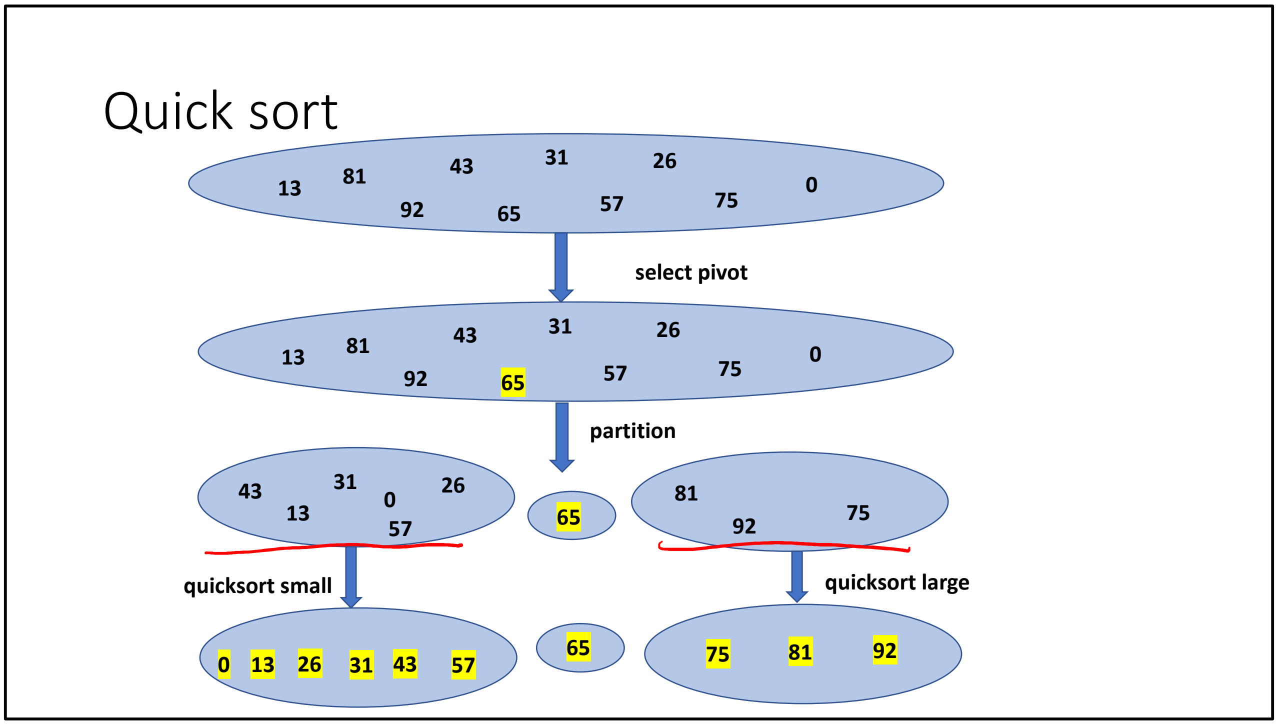
Quick sort

References

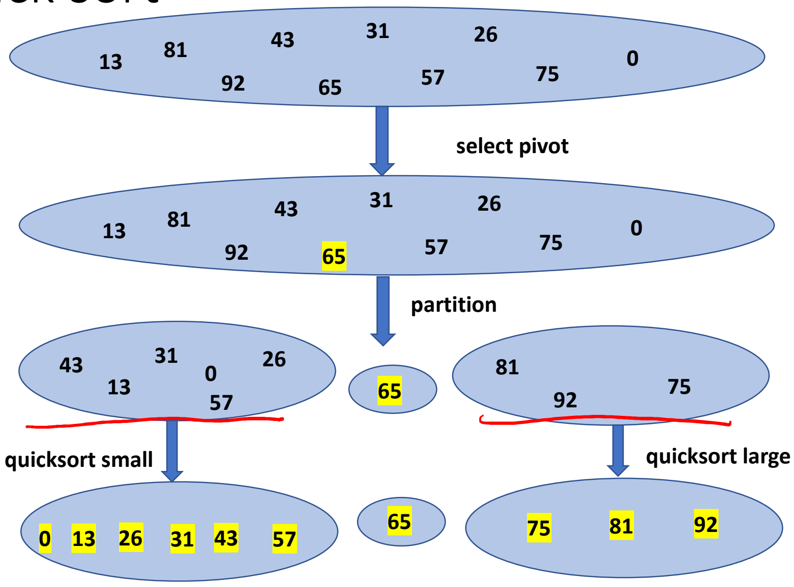
- Read section-7.7 from Mark Allen Weiss

Quick sort

- Quick sort is a recursive algorithm
- Base condition:
  - if the input array has one or zero elements, the array is already sorted; return the array
- Recursive step
  - Select an element `x` as pivot
  - Partition the array `A[lo .. hi]` into two sub-arrays `A[lo .. p-1]` and `A[p+1 .. hi]`
    - Every element in `A[lo .. p-1] ≤ A[p]`
    - Every element in `A[p+1 ] ≥ A[p]`
    - `A[p] = x     // so x is at its proper position in the array`
  - Recursively sort `A[lo .. p-1]` and `A[p+1 .. hi]`
  - No need to combine the results!



Quick sort

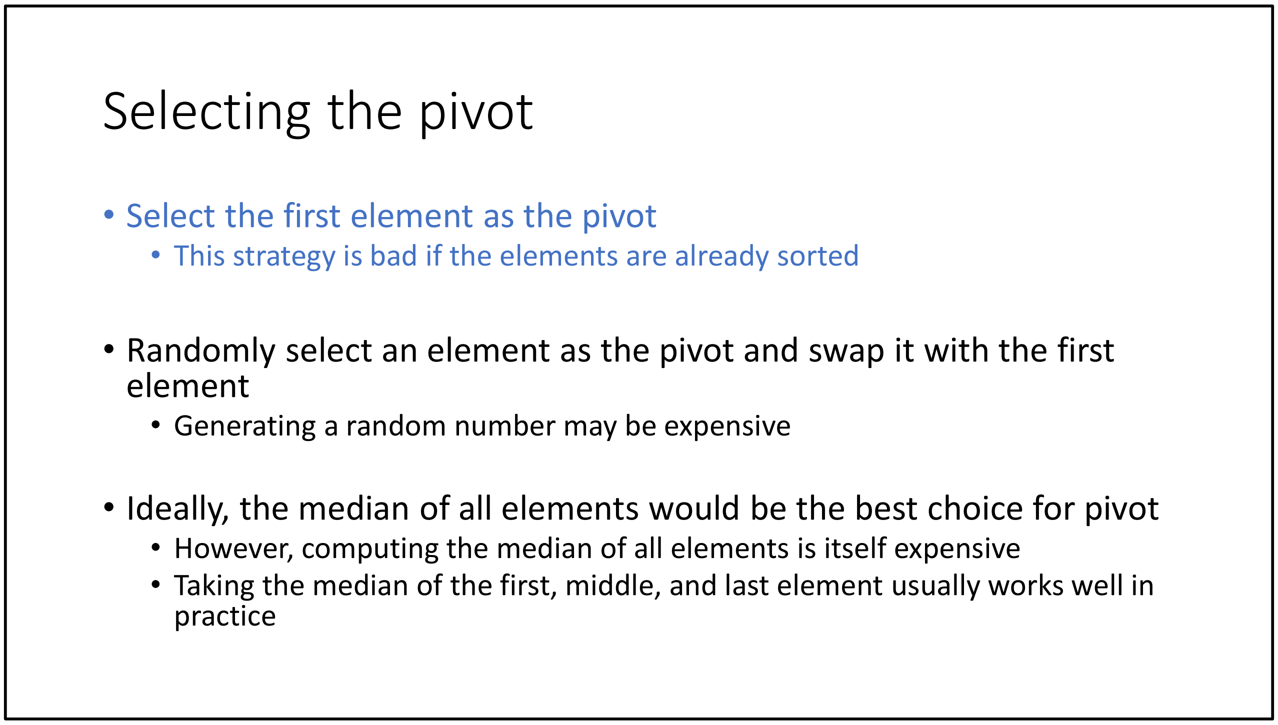


The heart of the quick sort algorithm is the partitioning algorithm. In the partitioning algorithm, we select an element in the array as the pivot and partition the array into two parts, left and right, so that all the elements in the left partition are less than the pivot and all the elements in the right partition are greater than the pivot. After the partitioning, the final position of the pivot is the same as in the sorted array. We never look at the pivot element again and recursively run the partitioning algorithm for the left and right subarray until all the elements in the array are stored at an index that is equal to their position in the sorted array.

Quick sort

- Quick sort algorithm performs best when the two partitions created after the partitioning are roughly of same size
- Pivot selection may affect the complexity of the quicksort algorithm

We will discuss later that the pivot selection is crucial for the performance of the quick sort algorithm. The quick sort performs best when the subarrays created after the partitioning are roughly of the same size. This could happen if we always select the median of n elements as the pivot. However, finding the median of n elements at low cost may be challenging.



Selecting the pivot

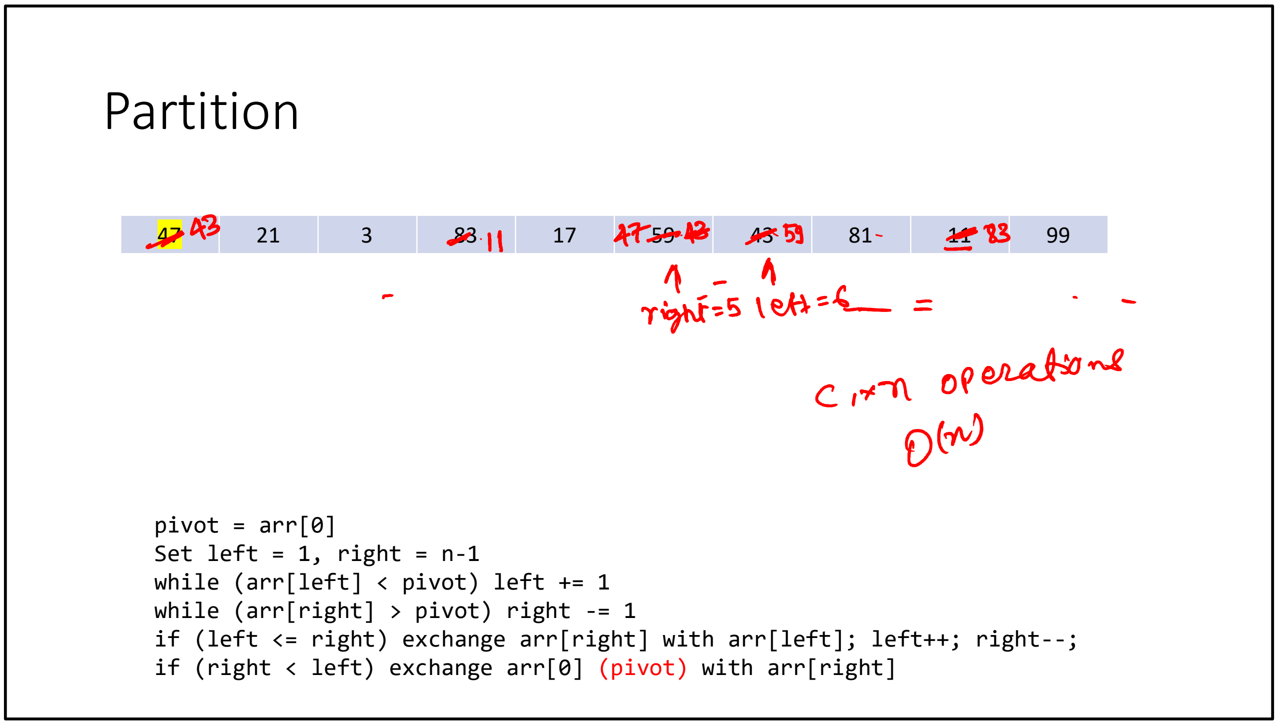
- Select the first element as the pivot
  - This strategy is bad if the elements are already sorted
- Randomly select an element as the pivot and swap it with the first element
  - Generating a random number may be expensive
- Ideally, the median of all elements would be the best choice for pivot
  - However, computing the median of all elements is itself expensive
  - Taking the median of the first, middle, and last element usually works well in practice

A simple strategy is to consider the first element as a pivot. However, this strategy is bad if the elements are already sorted, as we will see later.

Partition

47	21	3	83	17	59	43	81	11	99

```
The first element of the array is the pivot. You can partition the array
in multiple ways as long as the post-conditions for partitioning are met.
```



Partition

~~47~~ 43	21	3	~~83~~ 11	17	~~47~~ ~~59~~ 43	~~43~~ 59	81	~~11~~ 83	99

right=5 left=6

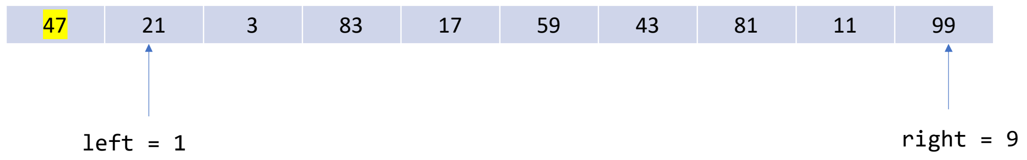
$c_1 \times n$ operations

$O(n)$

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

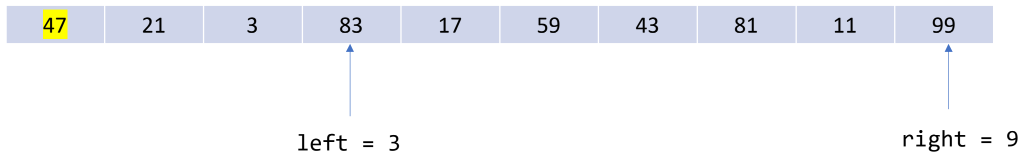
Let’s see how portioning is done. Initially, the first element is selected as the pivot. Now we use two variables, left and right, to keep track of elements that are less than or greater than the pivot. Initially, left is initialized to 1, and right is initialized to n-1, where n is the number of elements in the array. Now we keep incrementing left until we reach an index that contains a value greater than the pivot. Similarly, we keep decrementing the right until we reach an index that contains a value smaller than the pivot. Now, if left is greater than right, we know for sure that all the elements stored at indices greater than right are greater than the pivot, and all the values stored at indices less than or equal to right are less than the pivot. It means that the right holds the final position of the pivot, and we swap the pivot element with the element stored at the index right. However, if left is less than right, then we swap the values stored at indices left and right, increment the value of left, and decrement the value of right to ensure that all the elements stored before left are less than the pivot and all the elements stored after right are greater than the pivot. Notice that this algorithm will also work if there are duplicates in the array. This is something that you can explore yourself to verify. The following slides demonstrate the steps of this algorithm.

Partition



```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

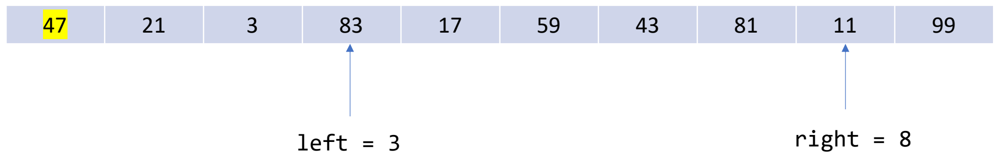
Partition



After while (arr[left] < pivot) left += 1

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

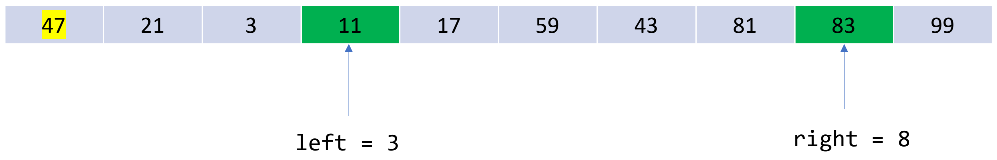
Partition



After while (arr[right] > pivot) right -= 1

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

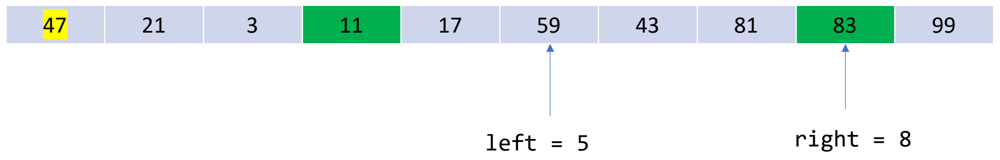
Partition



After if (left <= right) exchange arr[right] with arr[left]

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

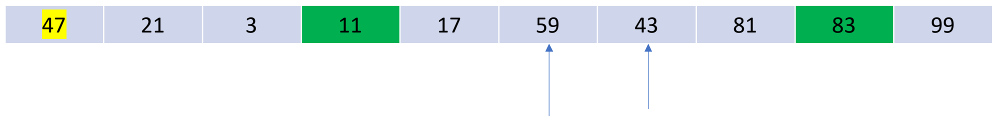
Partition



After while (arr[left] < pivot) left += 1

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

Partition

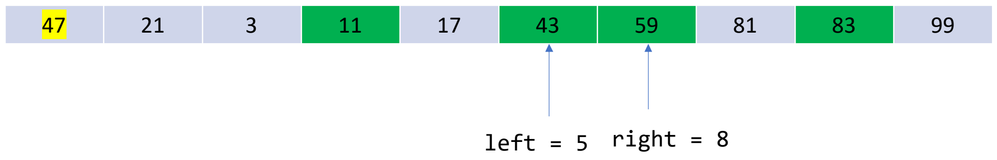


left = 5 right = 8

After while (arr[right] > pivot) right -= 1

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

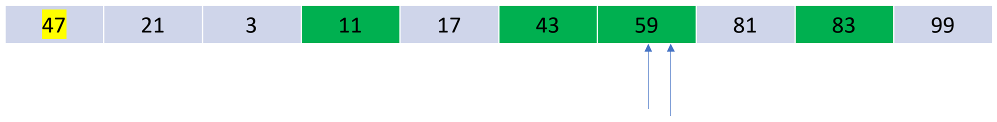
Partition



After (left <= right) exchange arr[right] with arr[left]

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

Partition

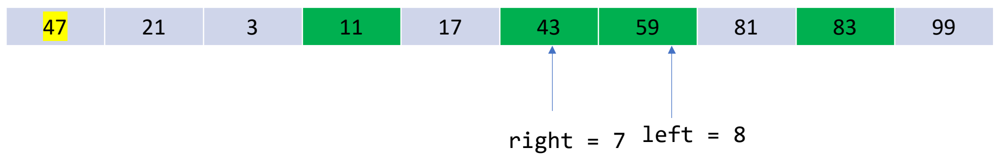


right = 8
left = 8

After while (arr[left] < pivot) left += 1

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

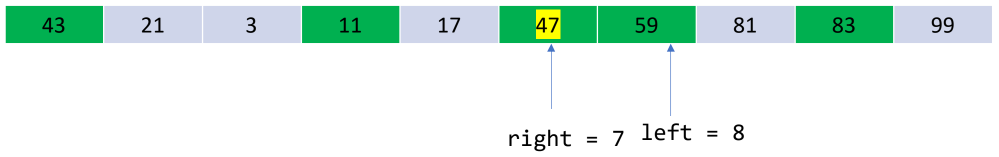
Partition



After while (arr[right] > pivot) right -= 1

```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

Partition



```
After if (right < left) exchange arr[0] (pivot) with arr[right]
```

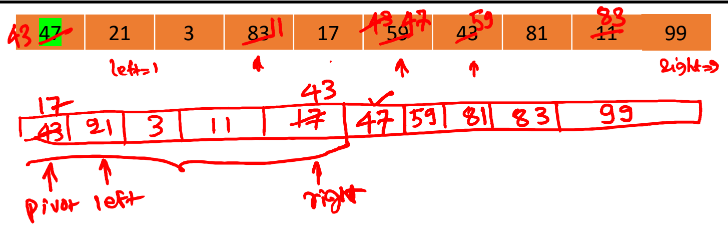
```
pivot = arr[0]
Set left = 1, right = n-1
while (arr[left] < pivot) left += 1
while (arr[right] > pivot) right -= 1
if (left <= right) exchange arr[right] with arr[left]; left++; right--;
if (right < left) exchange arr[0] (pivot) with arr[right]
```

Quick sort

- After partition, all the elements in the left subarray are less than the pivot, and all the elements in the right subarray are greater than pivot

- It means that after partition, the pivot is already at a position where it should be in the sorted array
  - No need to look at the final position of the pivot after partition!

- Recursively call quick sort on the left and right subarrays to place the remaining elements at their final plosions in the sorted array



Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at

The following slides show the state after every partition. The number highlighted in green shows the pivot before partition, the orange boxes are the scope of the current partitioning algorithm, and the yellow ones show the final position of the pivot after partitioning. Notice that we never look at an element again once we place it at its final destination.

47	21	3	83	17	59	43	81	11	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at

47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at

47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

AFTER PARTITION

17	21	3	11	43	47	59	81	83	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

AFTER PARTITION

17	21	3	11	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at

47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

AFTER PARTITION

17	21	3	11	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

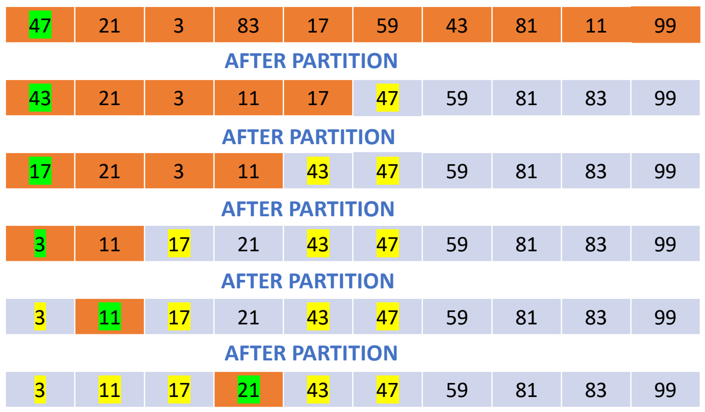
AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at



47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

AFTER PARTITION

17	21	3	11	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

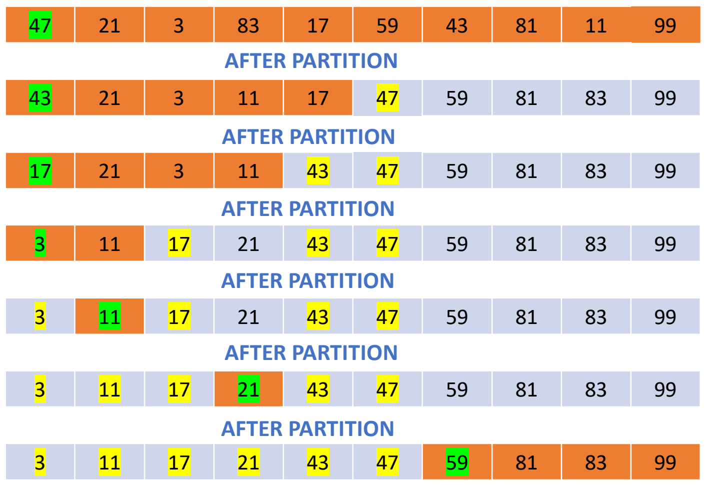
AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

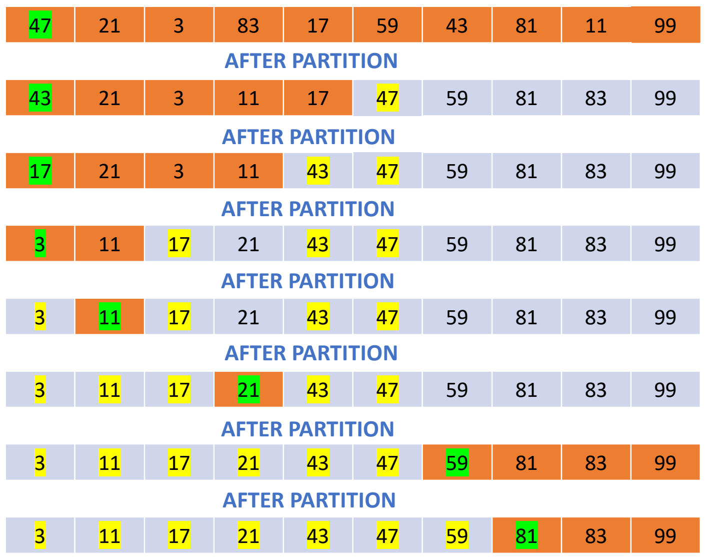
GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at



Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at



47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

AFTER PARTITION

17	21	3	11	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

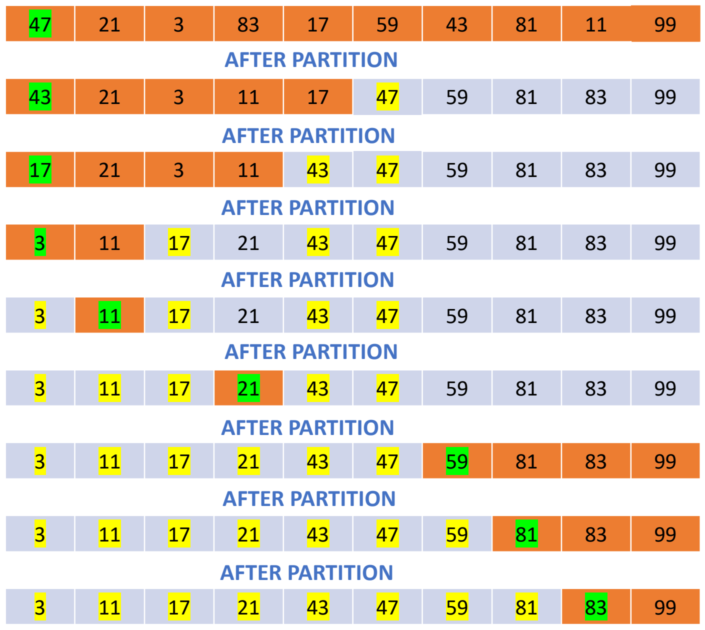
AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at



47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

AFTER PARTITION

17	21	3	11	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

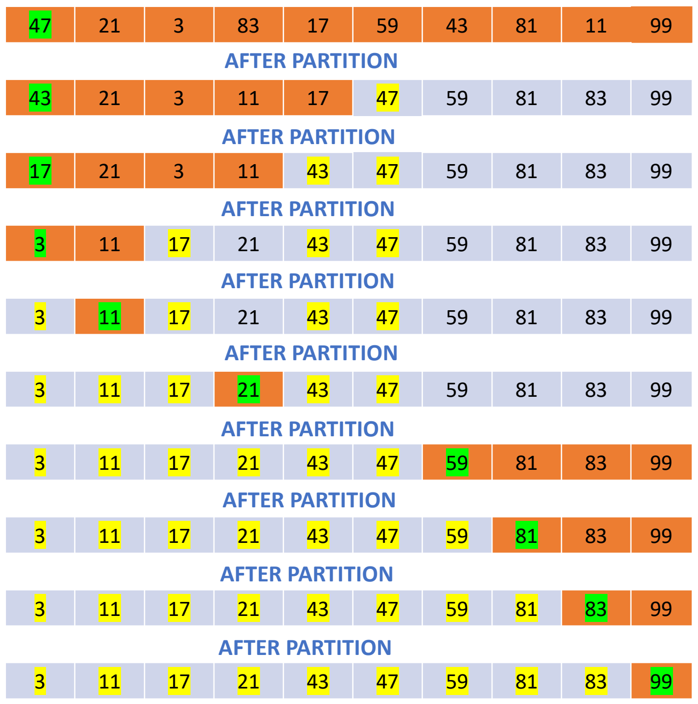
AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at



47	21	3	83	17	59	43	81	11	99

AFTER PARTITION

43	21	3	11	17	47	59	81	83	99

AFTER PARTITION

17	21	3	11	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

AFTER PARTITION

3	11	17	21	43	47	59	81	83	99

Quick sort

```
int p = partition(arr, lo, hi);
quicksort(arr, lo, p - 1);
quicksort(arr, p + 1, hi);
```

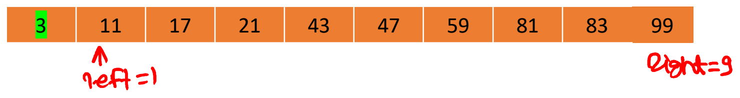
GREEN: initial position of pivot
YELLOW: final position of pivot
ORANGE: subarray we are looking at

Quick sort

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

```
int partition(int arr[], int lo, int hi)
{
  int pivot = arr[lo];
  int left = lo + 1;
  int right = hi;

  while (left <= right) {
    while (left <= right && arr[left] < pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left <= right) {
      exchange(arr, left, right);
      left++; right--;
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

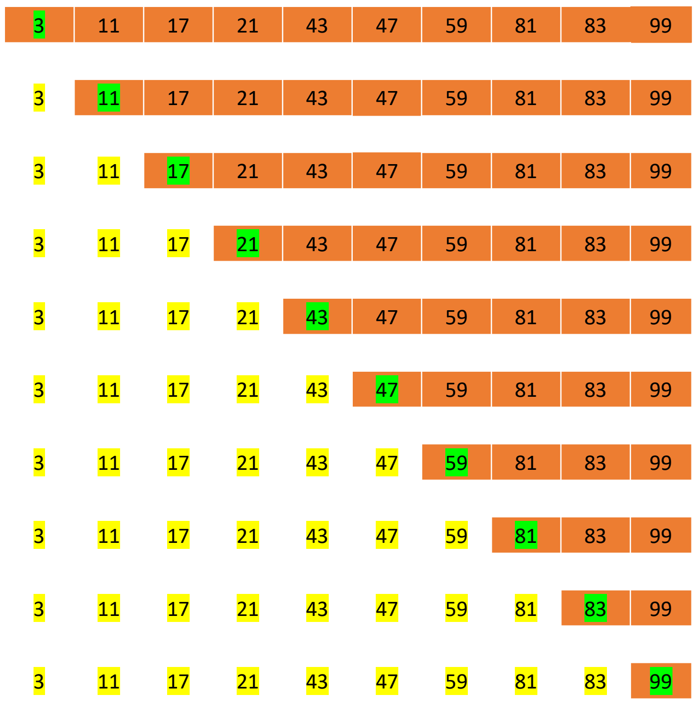
This is the source code of the quick sort algorithm.



Worst case partition

In the worst case, the partition algorithm generates two arrays of size zero and (n-1) for further sorting.

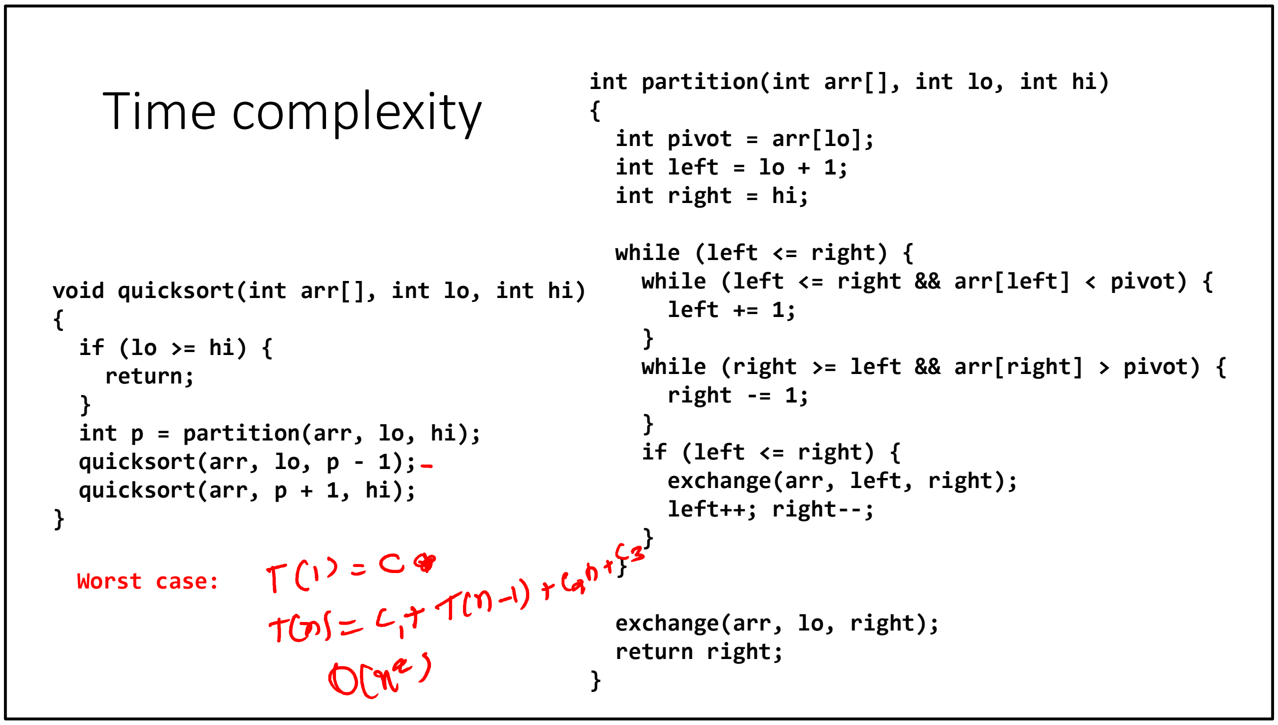
The worst-case partition results in two partitions of sizes zero and n-1. Notice that when that array is already sorted, we get the worst-case partition after every step.



Worst case partition

In the worst case, the partition algorithm generates two arrays of size zero and (n-1) for further sorting.

Time complexity



Time complexity

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);-
  quicksort(arr, p + 1, hi);
}
```

```
int partition(int arr[], int lo, int hi)
{
  int pivot = arr[lo];
  int left = lo + 1;
  int right = hi;

  while (left <= right) {
    while (left <= right && arr[left] < pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left <= right) {
      exchange(arr, left, right);
      left++; right--;
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

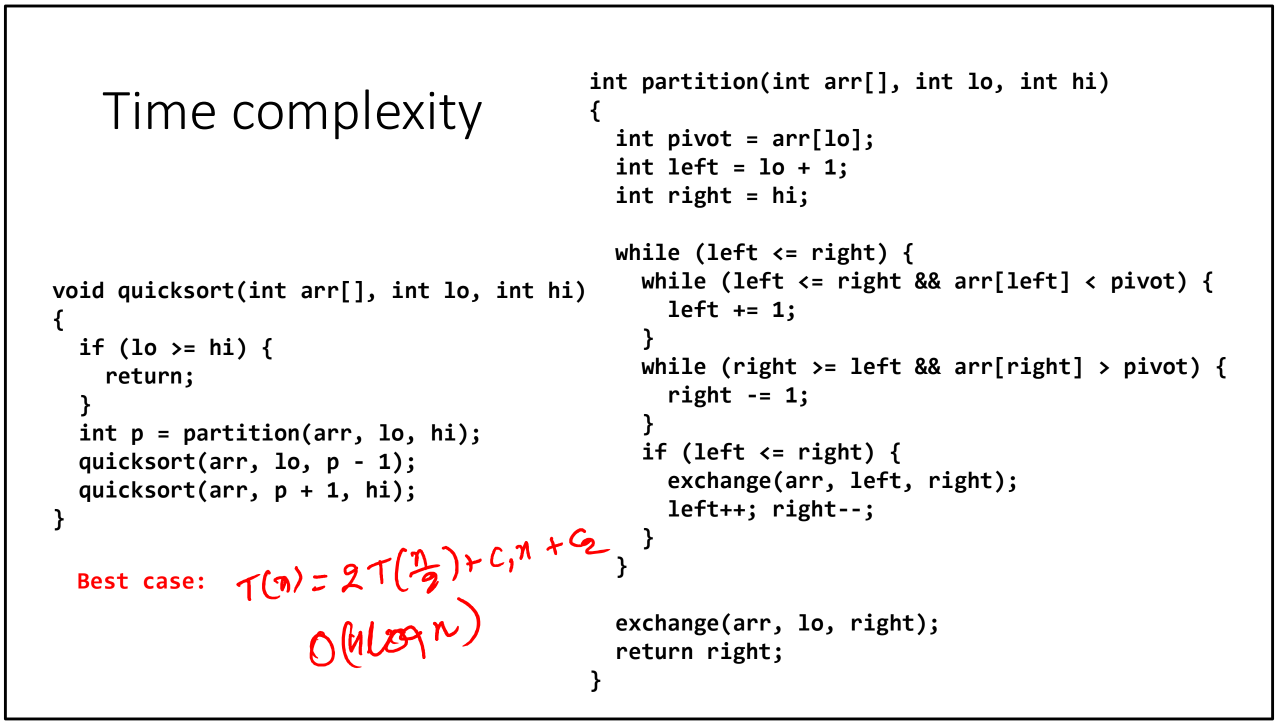
Worst case:

$T(1) = c_0$

$T(n) = c_1 + T(n-1) + c_2 n + c_3$

$O(n^2)$

In the worst case, the recursive call corresponding to no element performs constant operations. The other recursive call does T(n-1) operations. Also, the partition algorithm performs c_2 * n operations. The recurrence relation we get is similar to the recurrence relation for the selection sort algorithm. Therefore, the time complexity is $O(n^2)$.



Time complexity

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

```
int partition(int arr[], int lo, int hi)
{
  int pivot = arr[lo];
  int left = lo + 1;
  int right = hi;

  while (left <= right) {
    while (left <= right && arr[left] < pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left <= right) {
      exchange(arr, left, right);
      left++; right--;
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

Best case: $T(n) = 2T(\frac{n}{2}) + c_1 n + c_2$

$O(n \log n)$

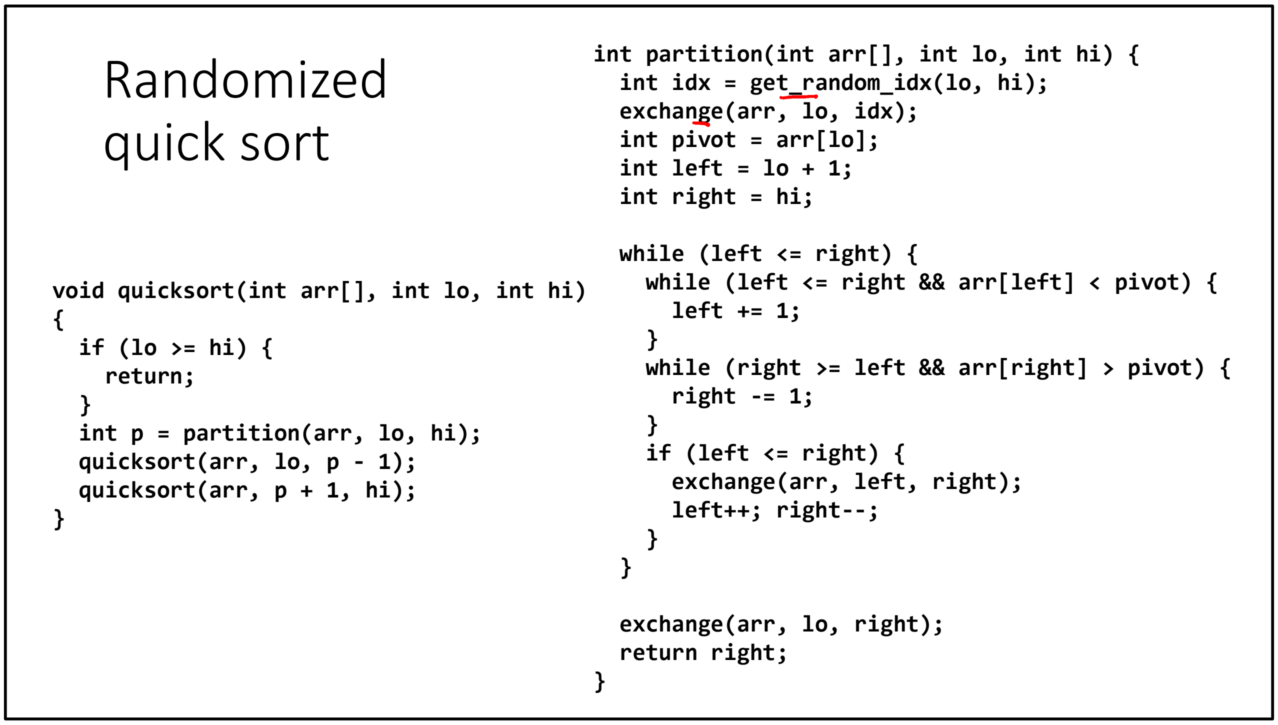
In the best case, we divide the array into two equal halves. In this case, the recurrence relation is similar to the one we obtained for the merge sort. Therefore, the best-case time complexity is O(n * log(n)).

Average case analysis

- Notice that the partitioning algorithm may divide an array in N different ways, where N is the size of the array
    - Possible partitions are: [0, N-1], [1, N-2], [2, N-3], …., [N-1, 0]
- If we pick the pivot in a truly random manner, the probabilities of each of these partitions would be 1/N,
    - Next slide shows the randomized quick sort algorithm

To compute the average time complexity, we need to ensure that all possible outcomes of the partitioning algorithm are equally likely. For this, we can use the randomized quick sort algorithm.

Randomized quick sort

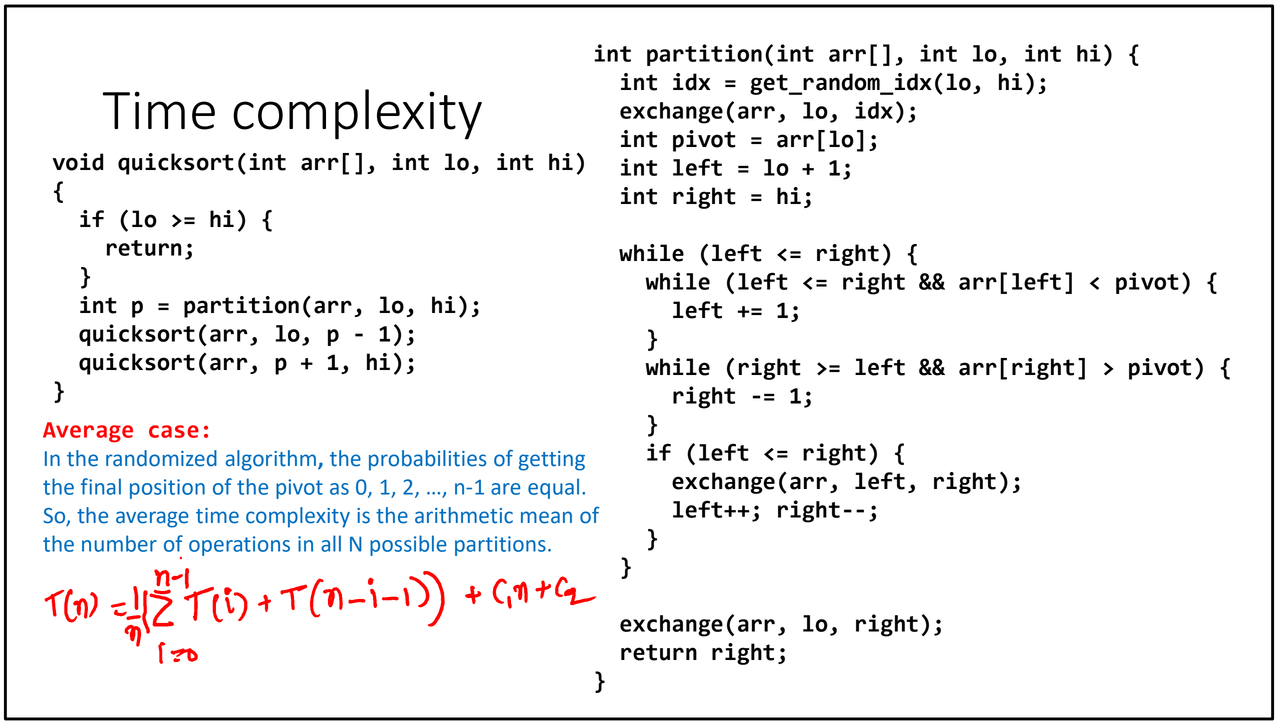


Randomized quick sort

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

```
int partition(int arr[], int lo, int hi) {
  int idx = get_random_idx(lo, hi);
  exchange(arr, lo, idx);
  int pivot = arr[lo];
  int left = lo + 1;
  int right = hi;

  while (left <= right) {
    while (left <= right && arr[left] < pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left <= right) {
      exchange(arr, left, right);
      left++; right--;
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

In the randomize quick sort algorithm, we pick the pivot element randomly. If the pivot is truly random, then the probability of getting the final position of the pivot as 0, 1, 2, …, N-1 would be 1/N. However, generating a truly random number is hard, and that is a separate discussion outside the scope of this course. This algorithm is suitable for average time complexity.



Time complexity

```
void quicksort(int arr[], int lo, int hi)
{
  if (lo >= hi) {
    return;
  }
  int p = partition(arr, lo, hi);
  quicksort(arr, lo, p - 1);
  quicksort(arr, p + 1, hi);
}
```

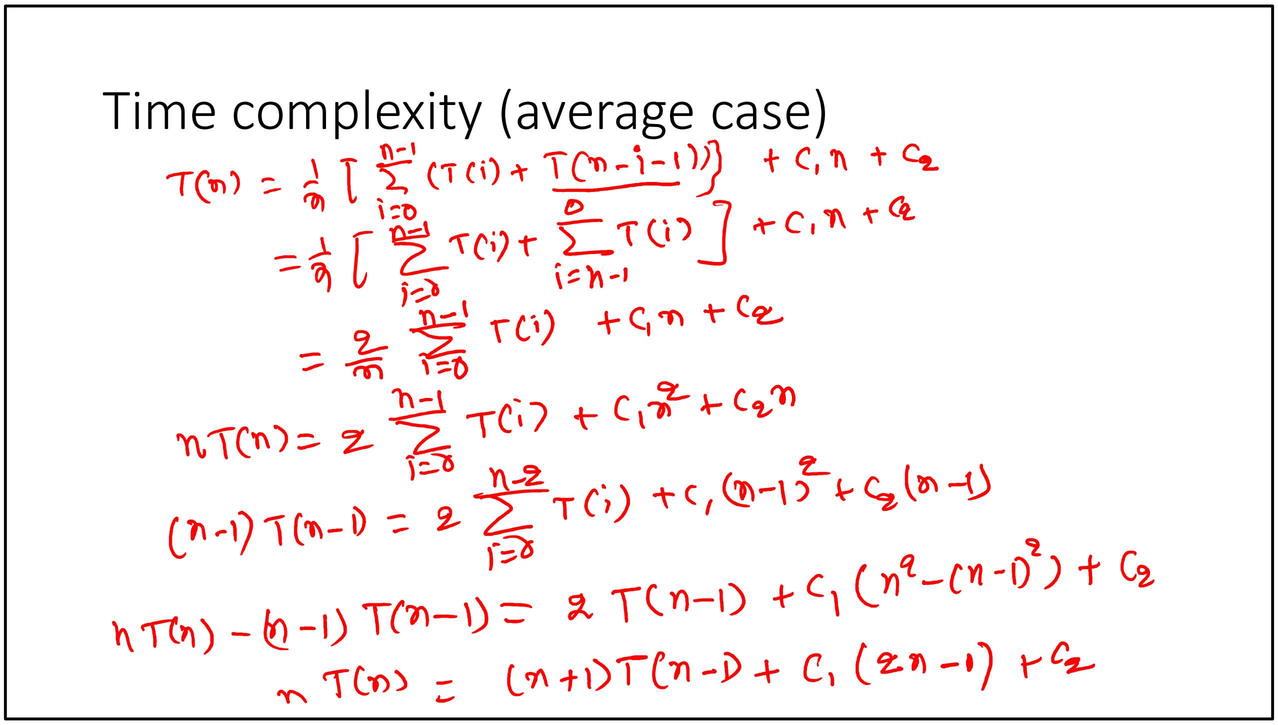
Average case:

In the randomized algorithm, the probabilities of getting the final position of the pivot as 0, 1, 2, …, n-1 are equal. So, the average time complexity is the arithmetic mean of the number of operations in all N possible partitions.

$$T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i) + T(n-i-1)\right) + c_1 n + c_2$$

```
int partition(int arr[], int lo, int hi) {
  int idx = get_random_idx(lo, hi);
  exchange(arr, lo, idx);
  int pivot = arr[lo];
  int left = lo + 1;
  int right = hi;

  while (left <= right) {
    while (left <= right && arr[left] < pivot) {
      left += 1;
    }
    while (right >= left && arr[right] > pivot) {
      right -= 1;
    }
    if (left <= right) {
      exchange(arr, left, right);
      left++; right--;
    }
  }

  exchange(arr, lo, right);
  return right;
}
```

If the partitioning algorithm returns an index i, the first recursive call performs T(i) operations and the second recursive call performs T(n-i-1) operations. If we take the average of all possible outcomes for i, we get the recurrence relation shown on this slide.



Time complexity (average case)

$$T(n) = \frac{1}{n}\left[\sum_{i=0}^{n-1}\left(T(i) + \underline{T(n-i-1)}\right)\right] + c_1 n + c_2$$

$$= \frac{1}{n}\left[\sum_{i=0}^{n-1} T(i) + \sum_{i=n-1}^{0} T(i)\right] + c_1 n + c_2$$

$$= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + c_1 n + c_2$$

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + c_1 n^2 + c_2 n$$

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c_1 (n-1)^2 + c_2 (n-1)$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + c_1\left(n^2 - (n-1)^2\right) + c_2$$

$$nT(n) = (n+1)T(n-1) + c_1(2n-1) + c_2$$

To solve the recurrence relations in which the entire history is present, we can compute the value of T(n-1) and subtract it from T(n). This will ensure that most of the history will be canceled out and the corresponding equation is in the form that can be solved using the expansion method.