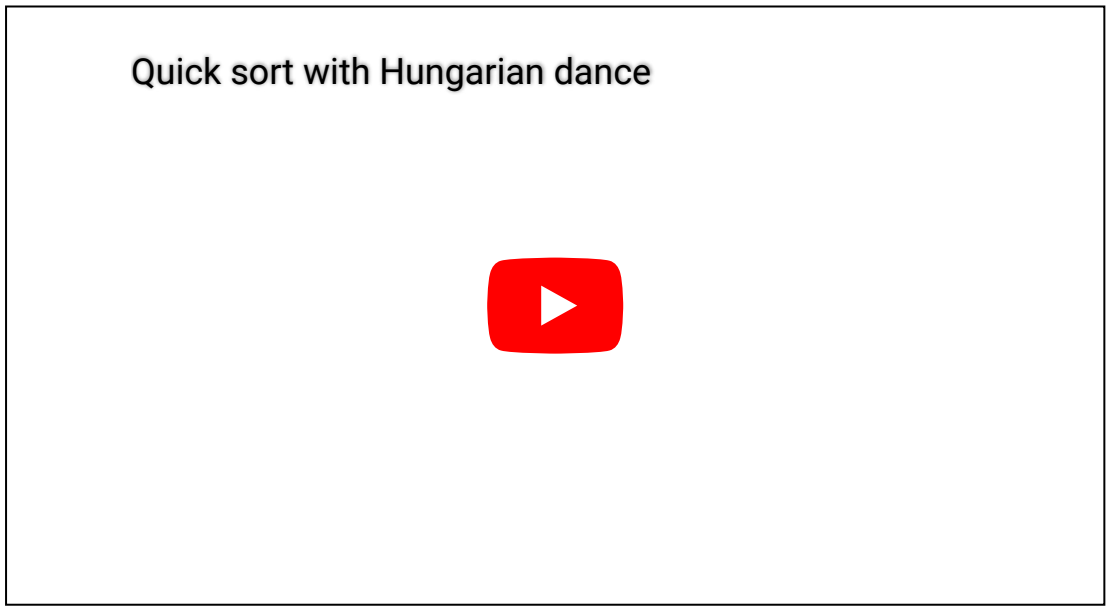
Merge sort

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists, until each sublist consists of a single element, merging then those sublists that will become a sorted list.



Quick Sorting

Like Merge Sort, Quick Sort is a Divide and Conquer algorithm, but even more efficient as we do not use the extra space to work with. It is the most commonly used algorithm in various programming languages. It picks an element that we call a pivot, then it rearranges them as such:

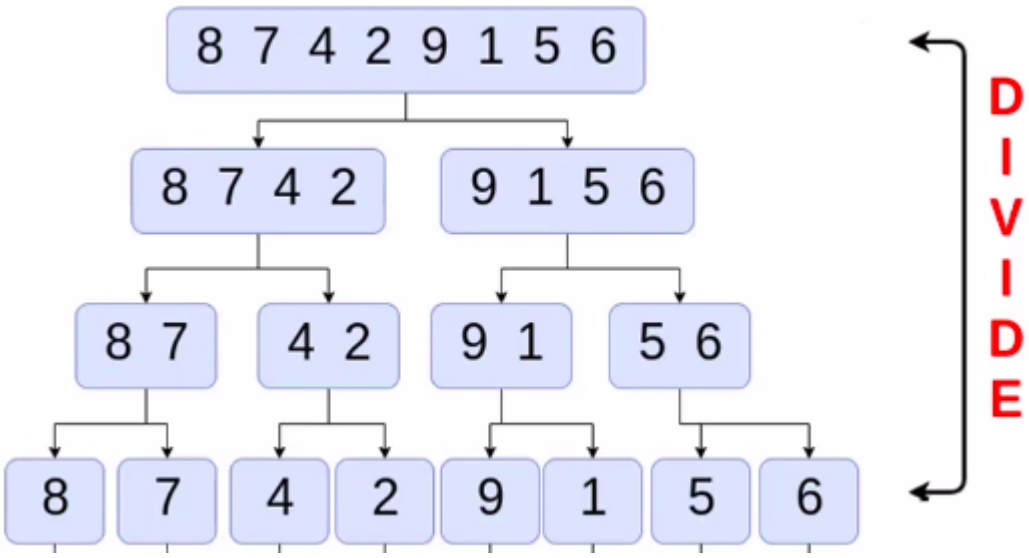
as the smallest items are pivoted to the left, and the largest items are pivoted to the right, this is what we call partitioning. The array is then divided into two partitions separated by a pivot.

It does not matter how the elements are ordered in each partition, what matters is positioning all smaller elements for the pivot element and all the largest elements right after that. A partition with only one element is considered a sorted partition.

1. Always pick last element as a pivot (shown in the example)
2. Always pick first element as a pivot.
3. Pick a random element as a pivot.
4. Pick median as a pivot.

Quick Sorting Algorithms.

Now let's see the implementation of the merge sort, but before let's take a look at this array in the image below:

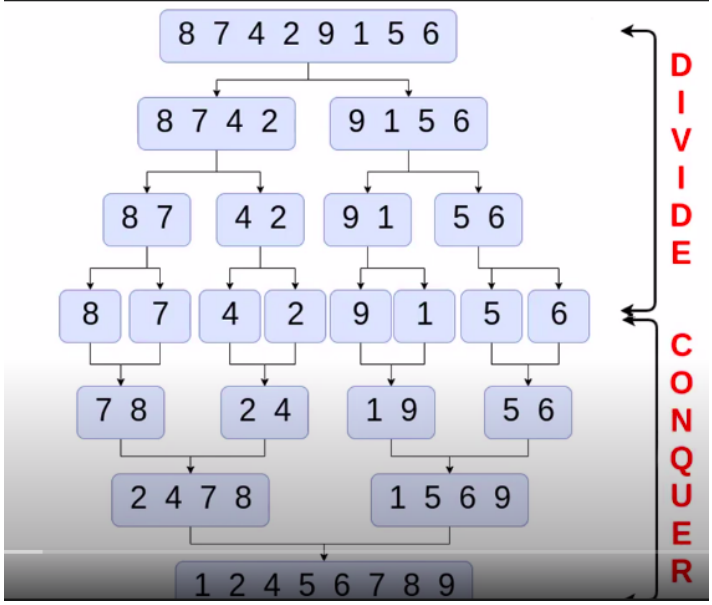


Do you remember the theory of divide and conquer, here we are going to use it again.

First, we are going to divide our array into smaller arrays to make sure that we get a single element in each array.

The next step in our algorithm is to merge each sub-array with one next to it in an ordered way.

We will take the image below into consideration :



Well, this is the implementation of the merge sort algorithm:

```
PROCEDURE merge_sort(VAR arr : ARRAY_OF INTEGER)
VAR
   i, m, mid, from, to, high, low : INTEGER;
BEGIN
   low := 0;
   high := arr.length-1;
   // divide the array into blocks of size m
   // m ={1,2,4,8,16,...}
   m := 1;
   WHILE (m<= high - low) DO
       // for m = 1; i={0,2,4,6,8,...}
       // for m = 2; i={0,4,8,...}
       // for m = 4; i={0,8,...}
       //....
       FOR i FROM low TO high-1 STEP 2*m  DO
           from := i;
           mid := i+m-1;
           to := min(i+2*m-1,high);
           merge(arr,from,mid,to);
       END_FOR
       m := 2*m;
   END_WHILE
```

```
END
```

This the merge procedure:

```
PROCEDURE merge(VAR arr : ARRAY_OF INTEGER, left, mid, right : INTEGER)
VAR
   i,j,k : INTEGER;
   n1 : INTEGER := mid - left + 1;
   n2 : INTEGER := right - mid;
   L : ARRAY_OF INTEGER[n1];
   R : ARRAY_OF INTEGER[n2];
BEGIN
   // copy data to temp arrays L[] and R[]
  FOR i FROM 0 TO n1-1   DO
      L[i] := arr[left+i];
  END_FOR
  FOR j FROM 0 TO n2-1 DO
      R[j] := arr[mid+1+j];
  END_FOR
  // Merge the temp arrays back into arr[left .. right]
  i := 0;
  j := 0;
  k := left;
  WHILE (i<n1 AND j<n2) DO
      IF (L[i] <= R[j]) THEN
          arr[k] := L[i];
          i := i+1;
       ELSE
          arr[k] := R[j];
          j := j+1;
      END_IF
      k := k+1;
  END_WHILE
```

```
  /* Copy the remaining elements of R[], if there are any */
  WHILE (j < n2) DO
      arr[k] := R[j];
      j := j+1;
      k := k+1;
  END_WHILE
END
```

This is the quick sort algorithm:

```
/*
arr -> Array to be sorted
l-> starting index
h-> ending index
*/

PROCEDURE quick_sort(VAR arr : ARRAY_OF INTEGER)
VAR
  // Create an auxiliary stack
  stack : STACK;
  p : INTEGER;
BEGIN
   // pushing initial values of l and h to stack
   stack.push(l);
   stack.push(h);
   // keep popping from the stack while is not empty
   WHILE (NOT stack.isEmpty()) DO
       // pop h and l
       h := stack.pop();
       l := stack.pop();
       // set pivot element at its correct position
       // in a sorted array
       p := partition(arr,l,h);
       // if there are elements on the right side of the pivot,
```

```
        // then push right side to stack
        IF (p+1 < h) THEN
            stack.push(p+1);
            stack.push(h);
        END_IF
    END_WHILE

END
```

And this is the implementation of the partition function:

```
/*
This function takes the last element as a pivot,
places its element at its correct position in sorted
array, and places all smaller (than the pivot) to left
of pivot and all the greater elements to right of pivot
*/

FUNCTION partition(arr : ARRAY_OF INTEGER, low, high : INTEGER) : INTEGER
VAR
    b,i,pivot : INTEGER;
BEGIN
  pivot := arr[high]; //pivot
  i := low-1; // index of smaller element
  FOR i FROM low TO high-1 DO
        // if the current element is smaller than the pivot
        IF (arr[i] < pivot) THEN
            i := i+1;
            swap(arr[i],arr[i])
        END_IF
  END_FOR
    swap(arr[b+1],arr[i])
```

```
    RETURN b+1 ;
END
```

Previous next