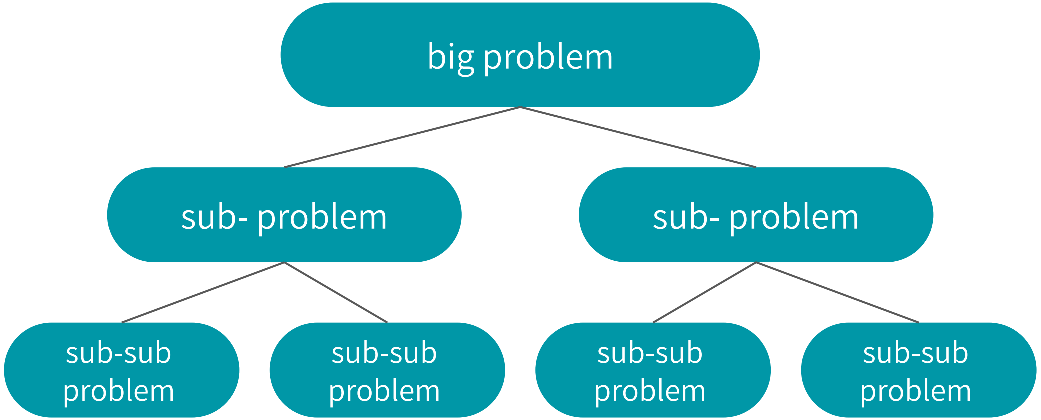
Lecture 17

Quick Sort
&
Merge Sort

October 21, 2021
Thursday

DIVIDE & CONQUER

- **DIVIDE-AND-CONQUER: an algorithm design paradigm**
  1. break up a problem into smaller subproblems
  2. solve those subproblems *recursively*
  3. combine the results of those subproblems to get the overall answer



Quick Sort

MOTIVATION

- Shell sort approached the problem of sorting
    - Dividing the original array into Subarrays.
    - Sorting them separately.
    - Dividing them again to sort new Subarrays.
    - Until the whole array was sorted.

- The goal was to reduce the original problem into subproblems that can be solved more ***quickly*** and easily.

QUICK SORT

- C. A. R. Hoare understood the principle well, and presented Quick Sort
- The original array is divided into two Subarrays.
  - First contains the items less than or equal to a chosen key called **pivot** or **bound.**
  - The second subarray contains the elements equal to or greater than the bound.
  - Two subarrays can be sorted separately, but before that partition process is repeated for both subarrays.

QUICK SORT

- Two new bounds are chosen, one for each subarray.
- The four subarrays are obtained, because the first two subarrays were divided into further two subarrays.
- The partitioning continues until there are only one-cell arrays that do not need to be sorted at all.

- This dividing process, it turns out the process of getting prepared to sort, the data have already been sorted.
- Quick sort is recursive in nature, because it is applied to both subarrays of an array at each level of partitioning.

PSEUDOCODE

```
QuickSort(array [ ] )
    if length ( array ) > 1
        choose bound;
        while there are elements left in array
            include element either in subarray₁ if e1 ≤ bound
            or in subarray₂ if e1 ≥ bound;
        QuickSort(subarray₁);
        QuickSort(subarray₂);
```

BOUND

- Two operations have to be performed
    - A bound has to be found.
    - Array has to be scanned to place the elements in the proper subarrays.
- Choosing a good bound is not trivial.
    - If an array contains the numbers from 1 to 100, and bound is 2.
- Choose the first element as bound.
- Choose the middle element as bound.
- Randomly generate a number between *first* and *last*. Good Random Generator may take additional time
- Choose a median of three elements.
    - First, middle and last.

WHAT ABOUT ELEMENT EQUAL TO BOUND

- The pseudocode does not clarify where to put the element equal to bound
  - Less than or equal to the bound than $subarray_1$
  - Greater than or equal to the bound than $subarray_2$
- The difference between the lengths of two subarrays should be minimal.
- One approach two minimize the difference
  - Place the largest element at the end of the array.
    - This prevents the index *lower* from going out of bound.
    - Could happen in first inner loop if the largest element becomes the bound.

TIME COMPLEXITY

- The worst case occurs if in each pass of Quicksort the smallest or the largest element is selected as bound.
- The partitions require $n - 2 + n - 3 + \ldots + 1$ comparisons and for each parition
  - Only bound is placed in the proper position.
  - Resulting in $\boldsymbol{O(n^2)}$.
- The best case occurs when the bound divides an array into two subarraysof approximately equal length *n/2.*
  - $n + 2\,n/2 + 4\,n/4 + 8\,n/8 + \ldots + n\,n\,/\,n = n\,(\log n + 1)$
  - $\boldsymbol{O\,(n \log n)}$

TIME COMPLEXITY

- We have to ask, is the average case more closer to the worst case or best case?
  - In most scenarios it will be ***O (n log n)***
- In the extreme case, the tree can be turned into linked list in which every non-leaf node has one child.
- This phenomenon is possible and prevents us calling the Quick Sort ideal.

LIMITATIONS

- In some cases, the Quicksort can be expected to be anything but quick.
- It is inappropriate to use quicksort for small arrays.
- Finding the best bound is difficult.

VISUALIZATION


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
0
1
2
3
6
4
5
8

Merge Sort

MOTIVATION

- The problem with Quicksort is that in worst case its complexity reaches ***O* (*n*²)** .
  - Because it is difficult to control the partitioning process.
  - Different process of choosing bounds make the process fairly regular, however there is no guarantee.
- Another strategy could be to make partitioning process as easy as possible.
  - Dividing the array into two halves.
  - This is the base of ***Mergesort.***
- Developed by John von Neumann, and was one of the first sorting algorithms used on Computer.

MERGE SORT

- The process of dividing arrays into two halves stops when the array has fewer than two elements.
- The key process in merge sort is merging sorted halves of an array into one sorted array.
- The algorithm is recursive in nature.

PSEUDOCODE

```
MergeSort (data [ ] )
    if data have at least two elements
        MergeSort( left half of data ) ;
        MergeSort( right half of data ) ;
        Merge (both halves into a sorted list );
```

PSEUDOCODE

```
Merge (array1 [ ], array2 [ ], array3 [ ] )
        i1, i2, i3 are properly initialized
                while both array2 and array3 contain elements
                        if array2 [ i2 ] < array3 [ i3 ]
                                array1 [ i1++ ]  = array2 [ i2++ ] ;
                        else array1 [ i1++ ]  = array3 [ i3++ ] ;
                load into array1 the remaining elements of either array2 or array3;
```

MERGE SORT

- The pseudocode for ***Merge*** suggests that array1, array2 and array3 are physically separate entities.
  - If we consider the array1 concatenation of array2 and array3 as before Merge.
  - We will end up with duplicate values.
  - For example: array2 [1 4 6 8 10] and array3 = [2 3 5 22];
  - If array1 is [1 4 6 8 10 2 3 5 22]
  - Then after second iteration of the loop array2 = [1 2 6 8 10]
    - and array1 = [1 2 6 8 10 2 3 5 22]
- Therefore a temporary array has to be used.

PSEUDOCODE

```
Merge (array1 [ ], first, second )
    mid = ( first + second ) / 2;
    i1 = 0; i 2 = first, i3 = mid + 1;
    while both left and right subarrays of array1 contain elements
            if array1 [ i2 ] < array1 [ i3 ]
                temp [ i1++ ]  = array1 [ i2++ ] ;
            else temp [ i1++ ]  = array1 [ i3++ ] ;
        load into temp the remaining elements of array1;
        load to array1 the contents of temp;
```

MERGE SORT

- Number of movements in each execution of merge ( ) is always the same and equal to 2 (last - first + 1).
- The number of comparisons depends on the ordering of array1.
  - If array1 is inorder or the elements in the right half precede th elements in the left half
    - number of comparisons is (first + last) / 2.

MERGE SORT

- The worst case is when the last element of one half precedes the last element of the other half.
  - In this case the number of comparisons = last - first.
- For n-elements array the number of comparisons is n - 1

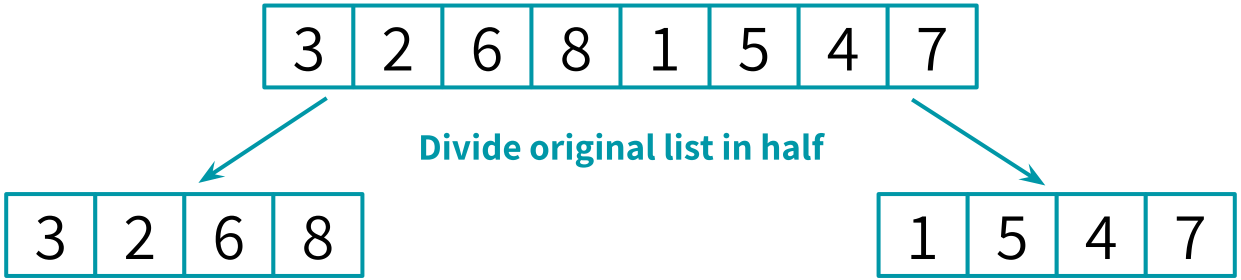
PSEUDOCODE

```
MergeSort (data [ ], first, second )
    if first < last
        mid = ( first + last )  / 2 ;
        MergeSort ( data, first, mid);
        MergeSort (data, mid+1, last);
        Merge (data, first, last);
```

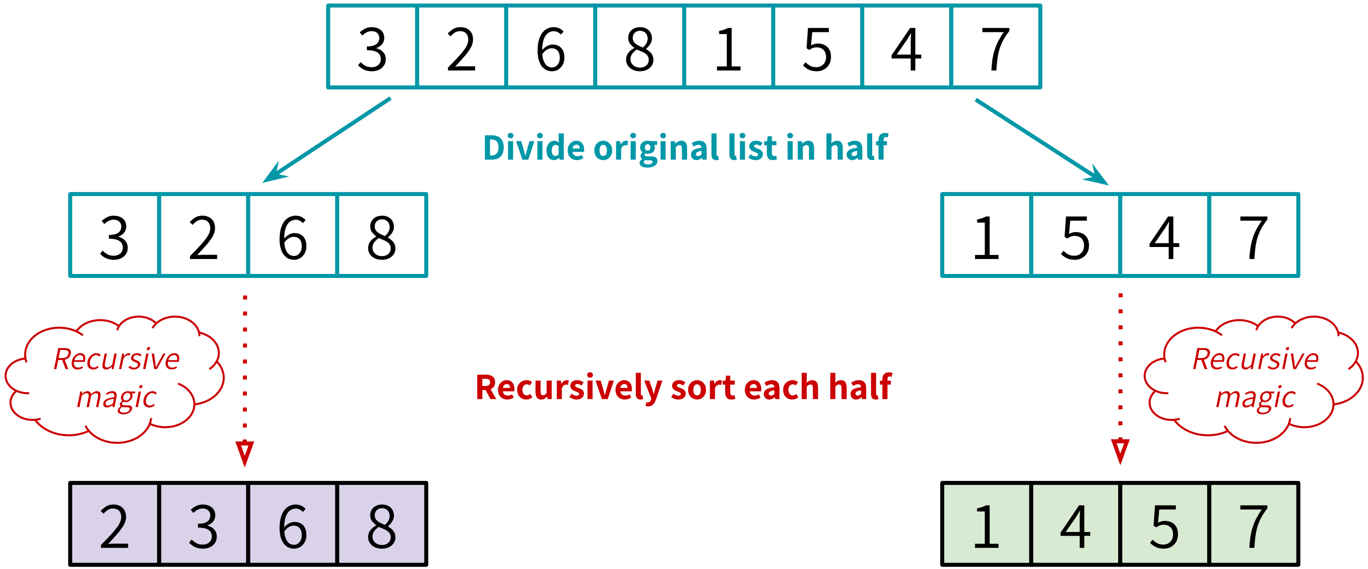
MERGESORT

3	2	6	8	1	5	4	7

MERGESORT

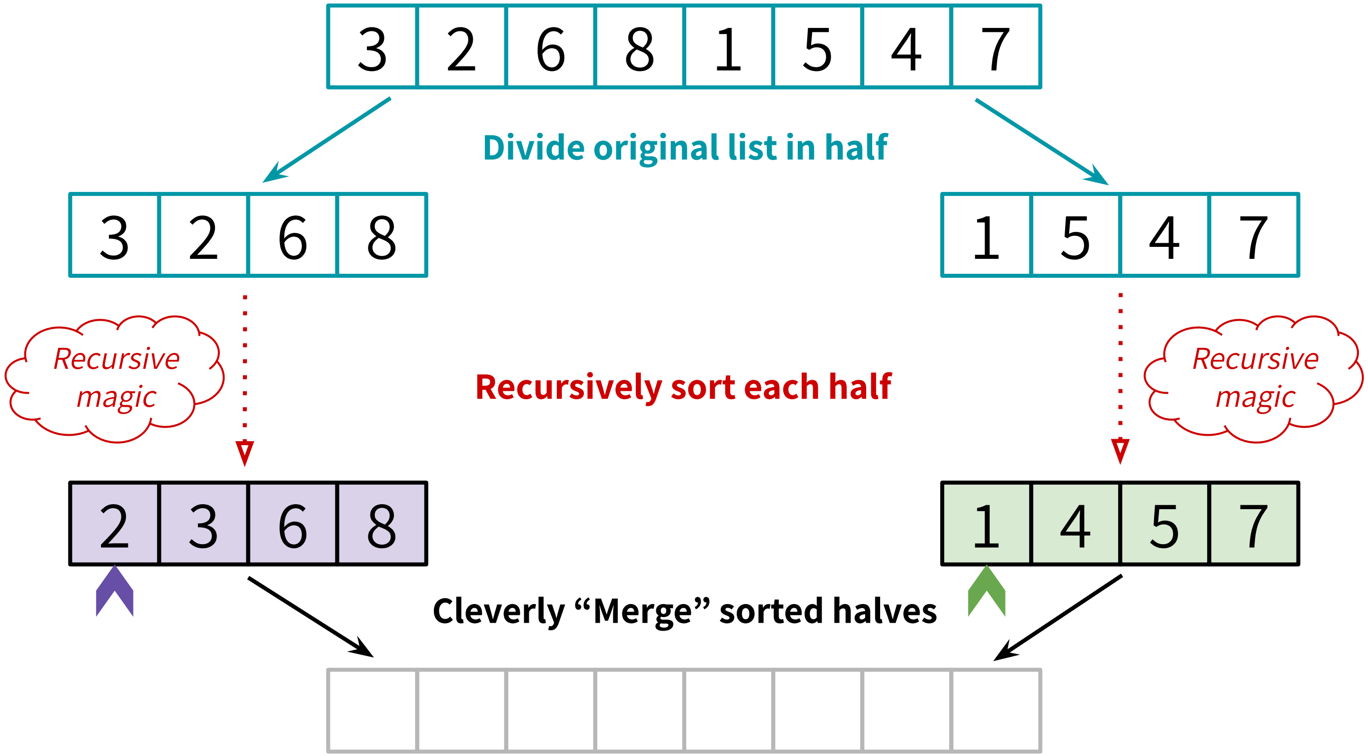


MERGESORT

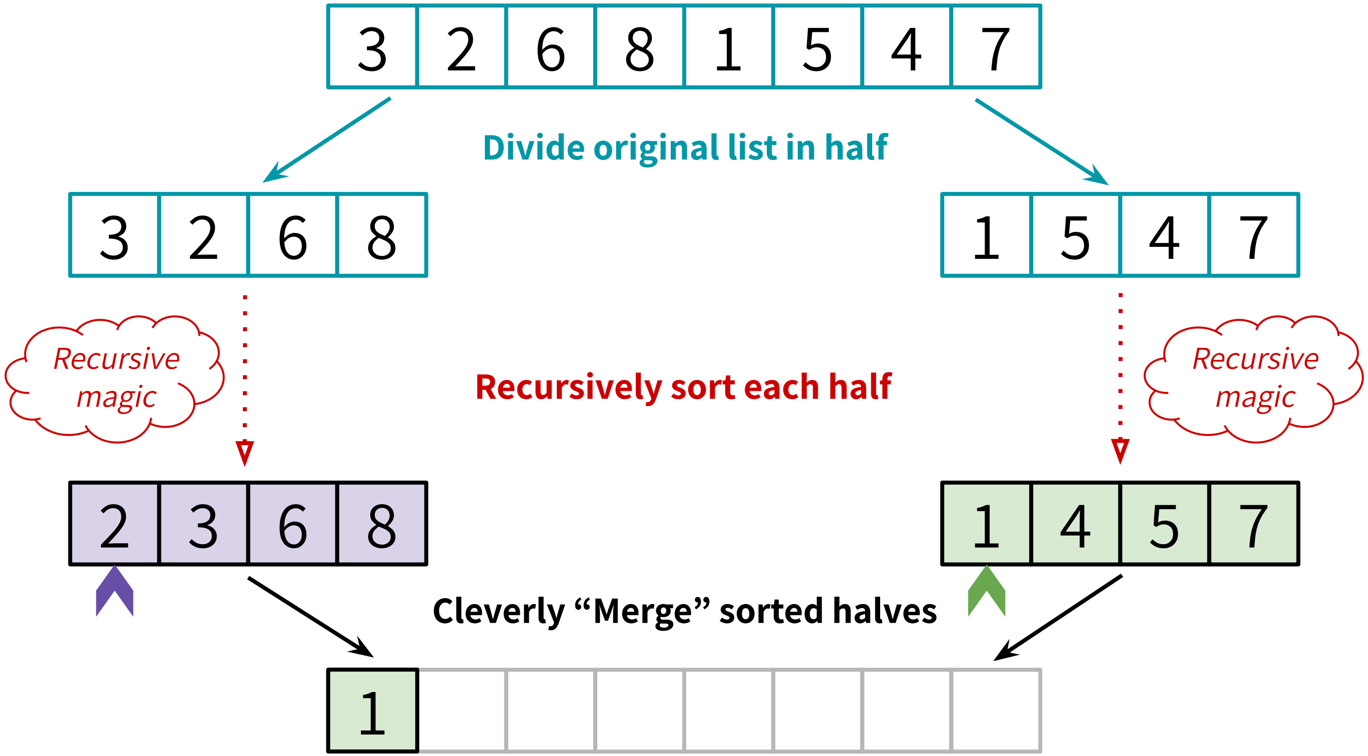


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7

MERGESORT

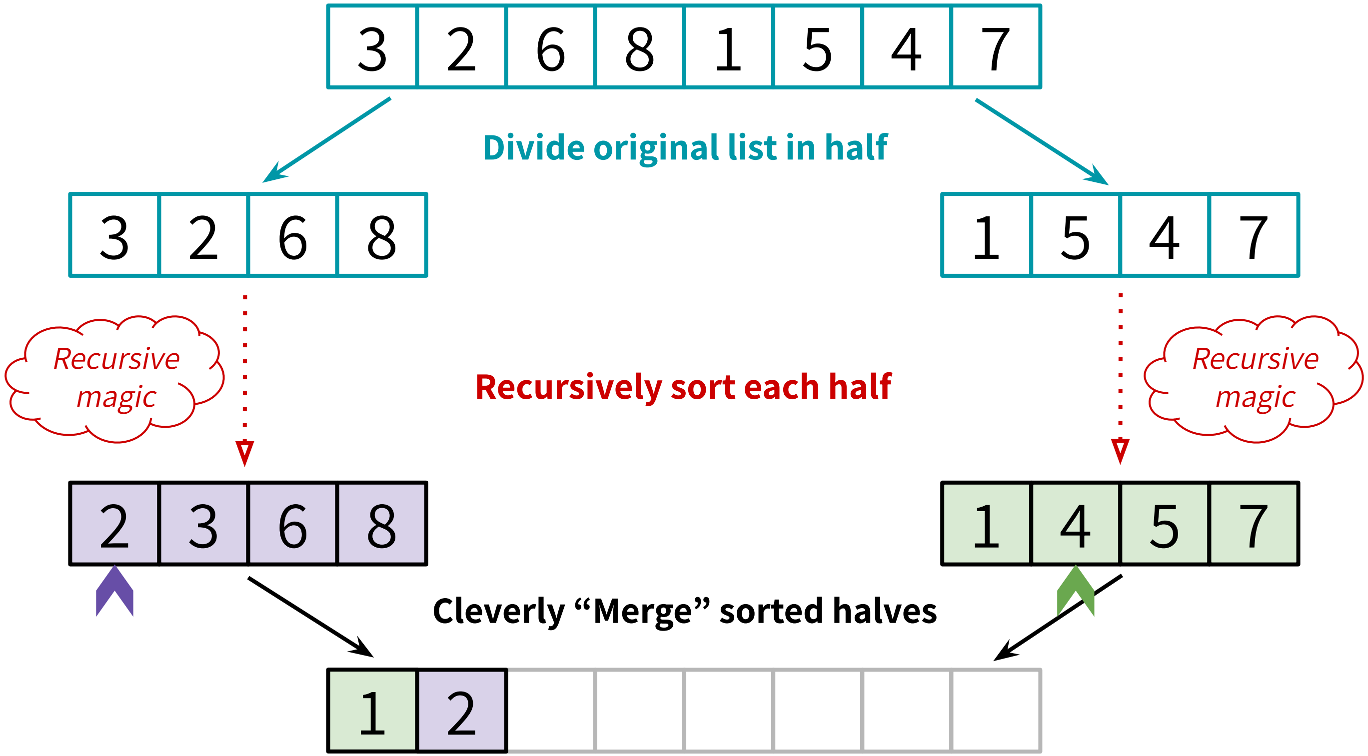

3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves

MERGESORT

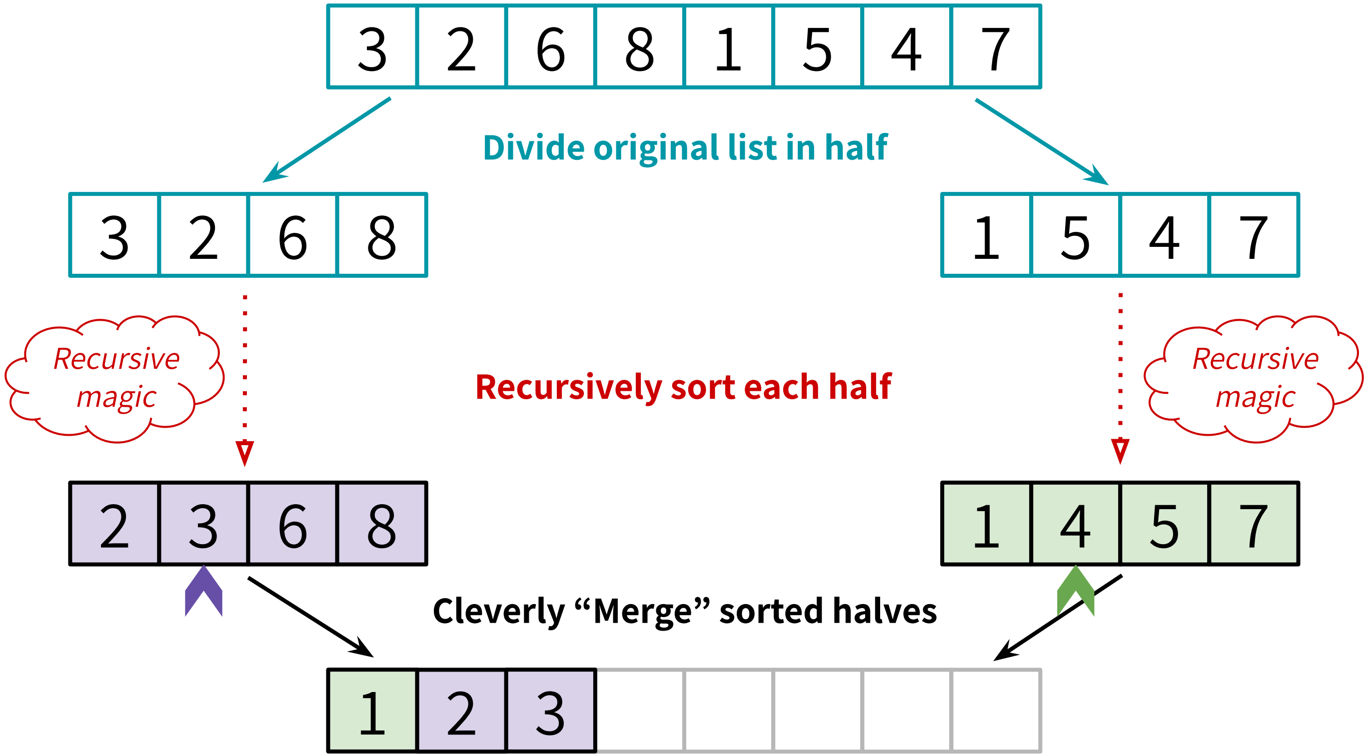


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1

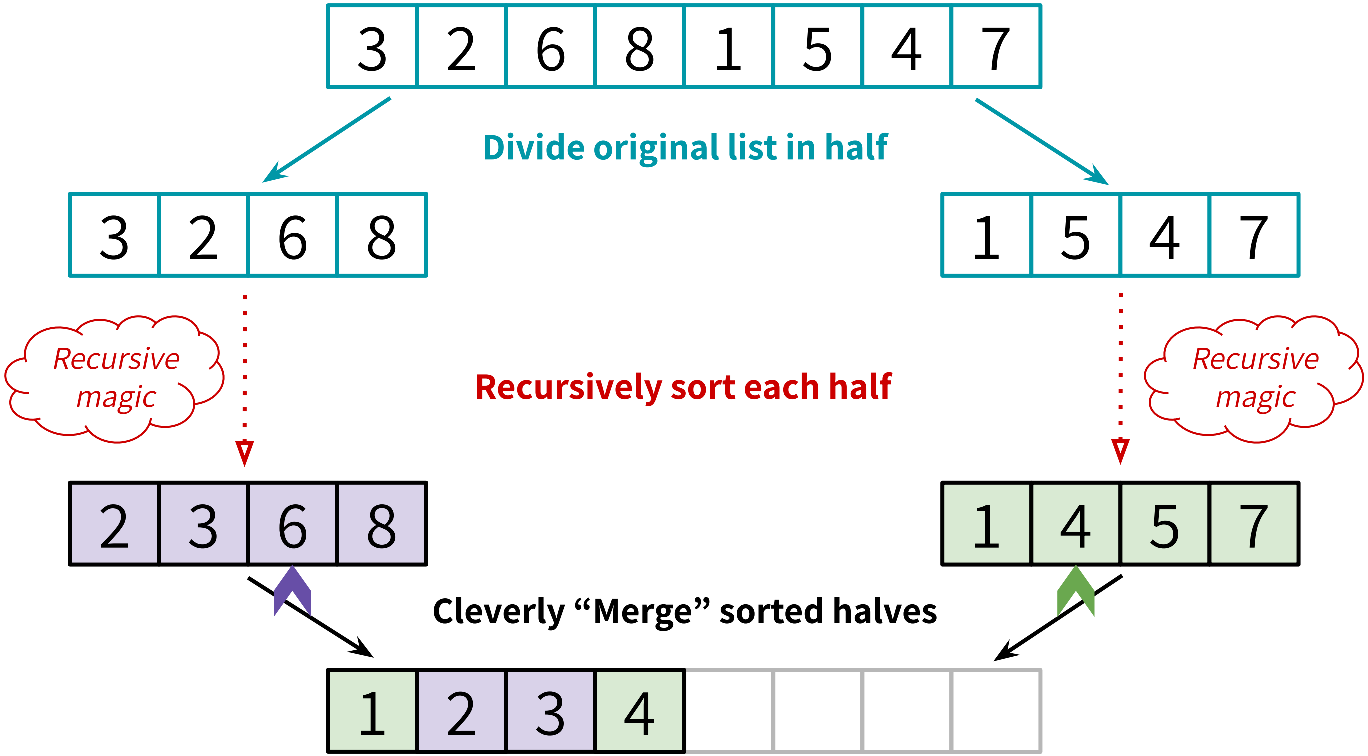
MERGESORT


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2

MERGESORT

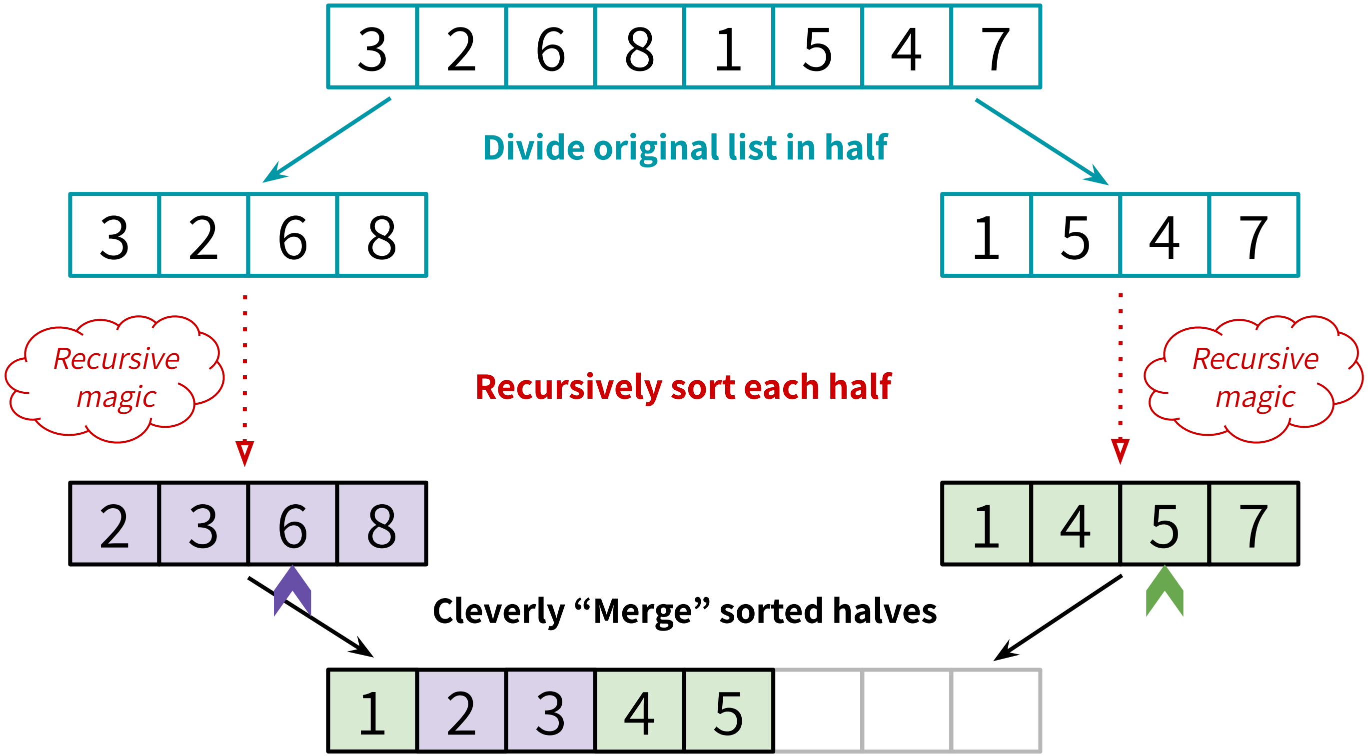

3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3

MERGESORT

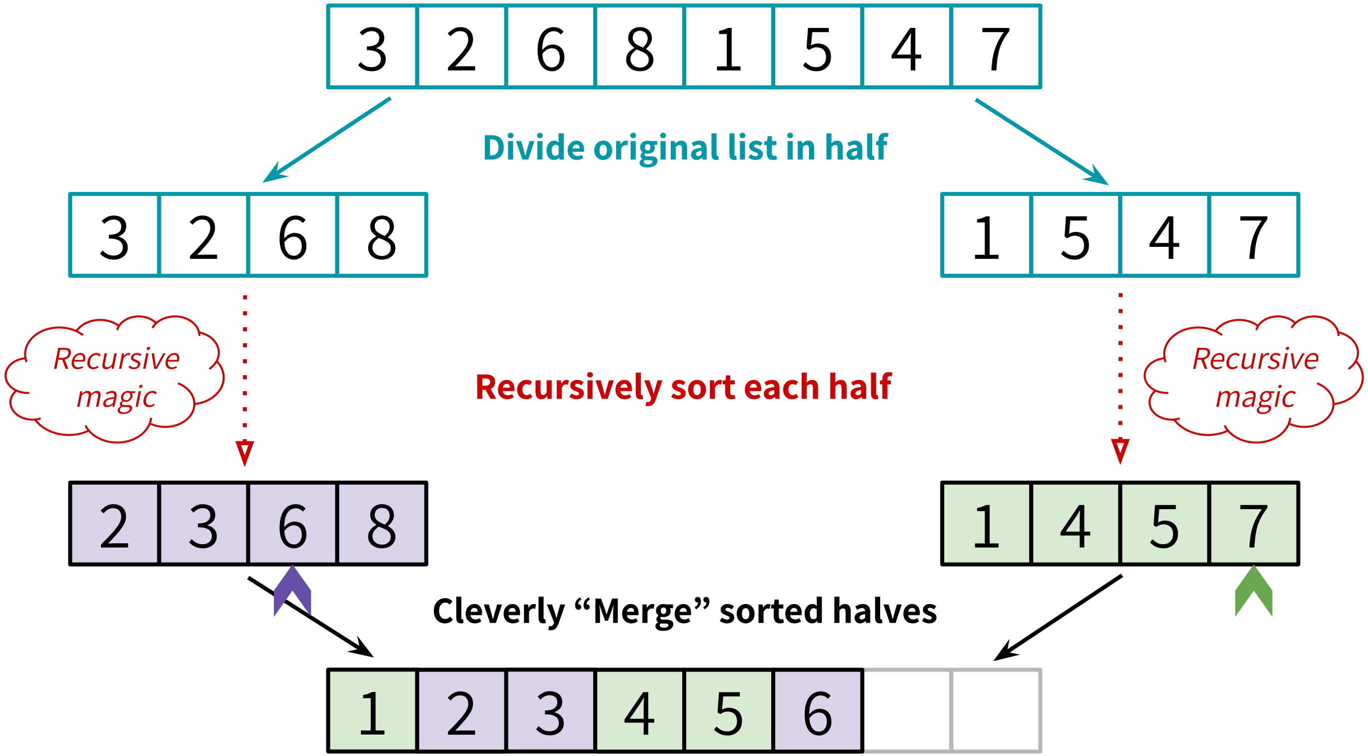


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4

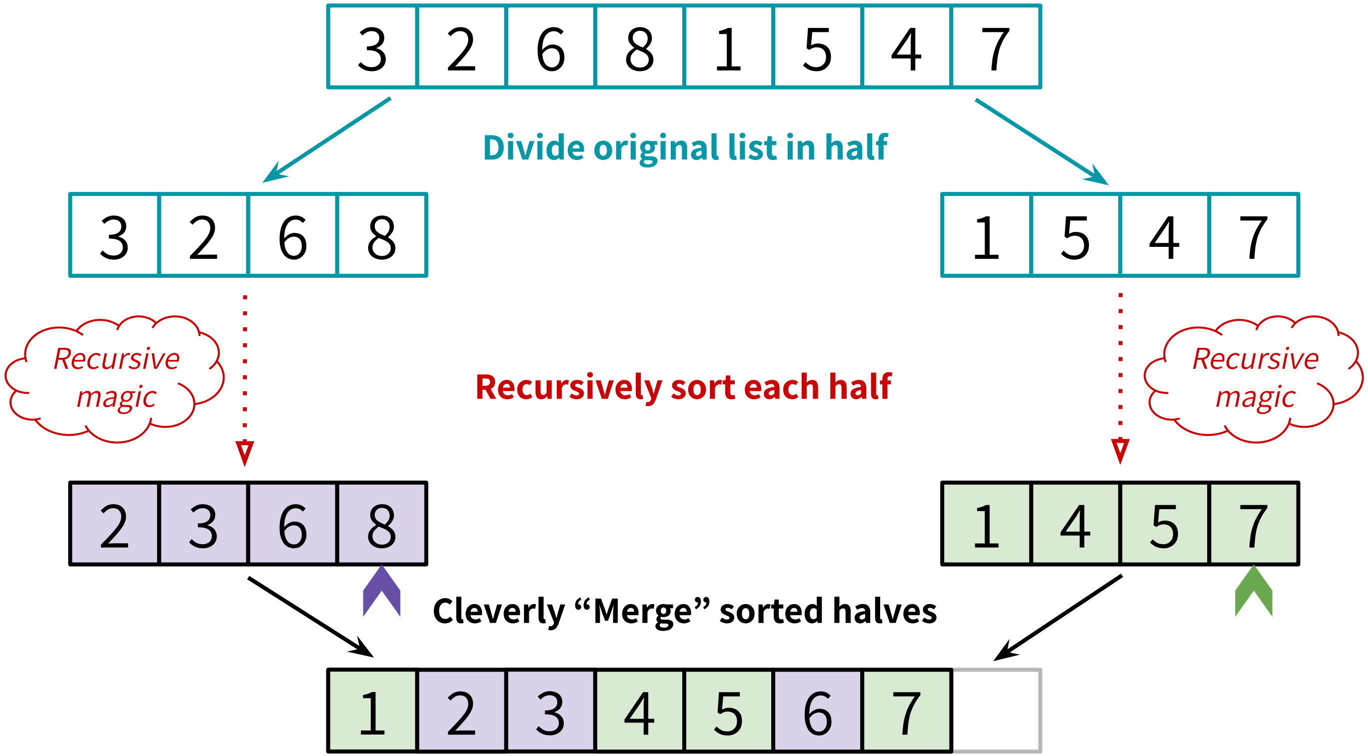
MERGESORT


3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2 3 4 5

MERGESORT

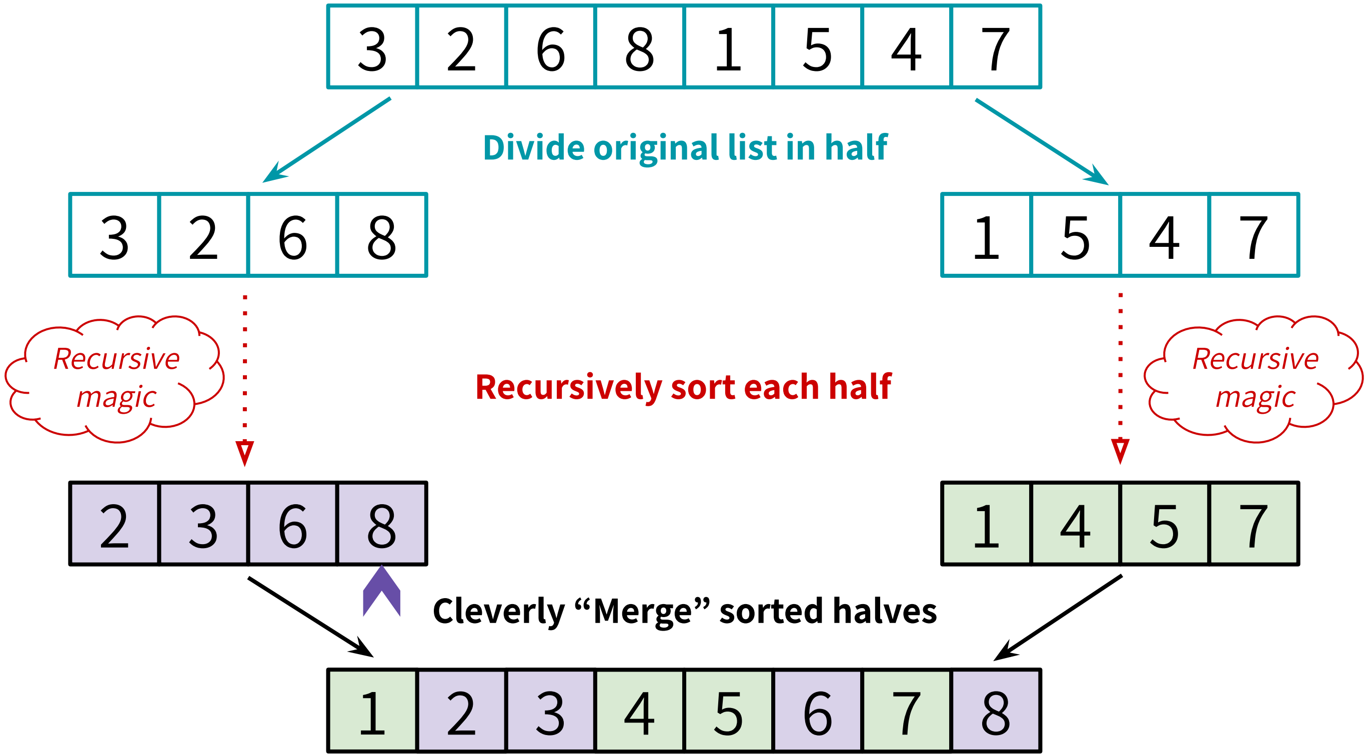

3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5 6

MERGESORT



3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly “Merge” sorted halves
1 2 3 4 5 6 7

MERGESORT



3 2 6 8 1 5 4 7
Divide original list in half
3 2 6 8
1 5 4 7
Recursive magic
Recursively sort each half
Recursive magic
2 3 6 8
1 4 5 7
Cleverly "Merge" sorted halves
1 2 3 4 5 6 7 8

TIME COMPLEXITY & ISSUES

- In all cases the Merge sort complexity is ***O (n log n)***
- However, the major drawback of Merge Sort is
  - The requirement of additional storage for merging the data
  - Which for large amounts of data can be insurmountable obstacle.

VISUALIZATION


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
2
4
0
6
8
5
1
7
3
9
b[0]
b[1]
b[2]
b[3]
b[4]
b[5]
b[6]
b[7]
b[8]
b[9]