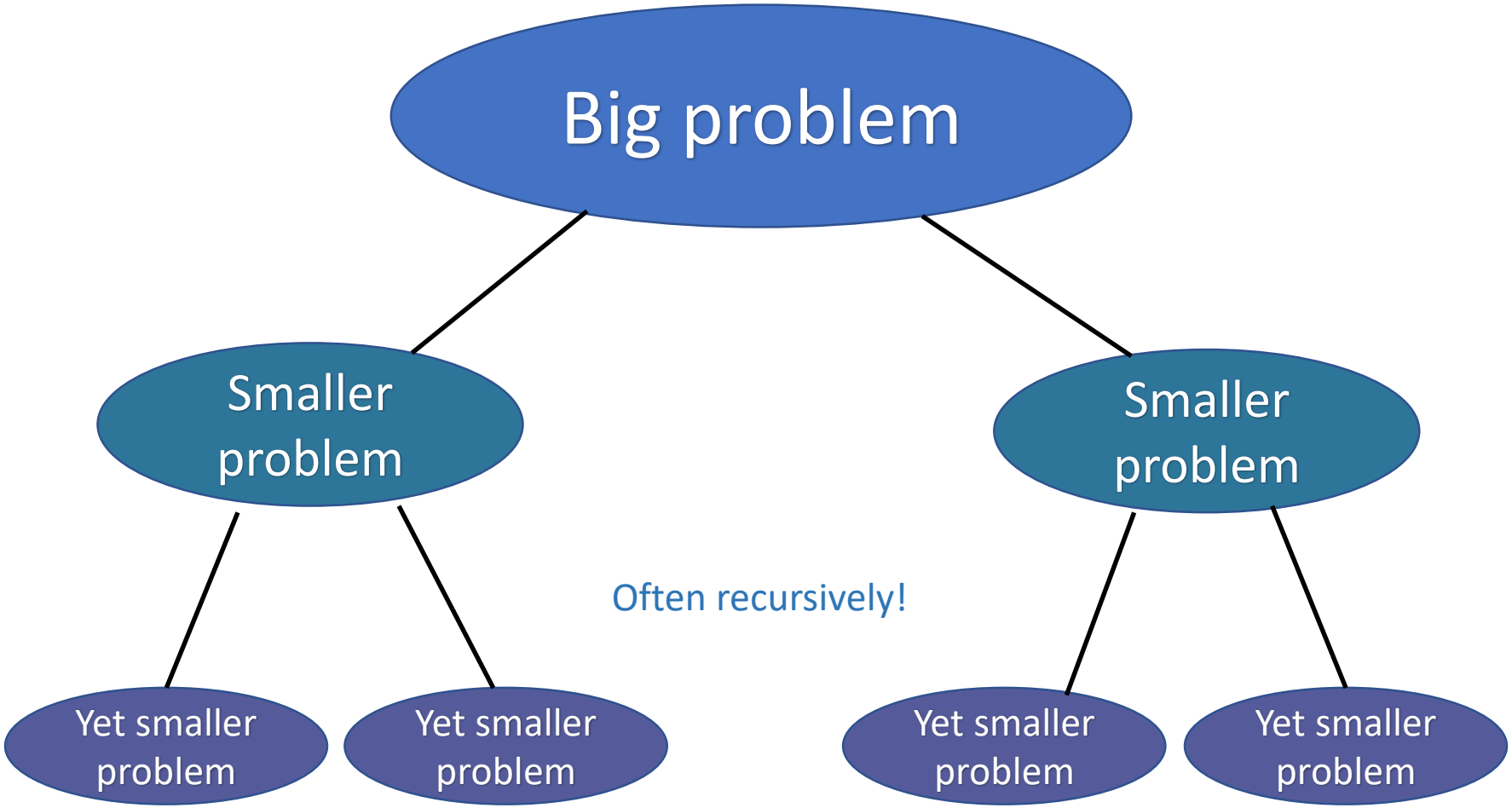
Divide and Conquer

Divide and conquer

- Break problem up into smaller (easier) sub-problems



http://web.stanford.edu/class/archive/cs/cs161/cs161.1182/

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.

- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.

- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.

- A comparison between the sorting algorithm is important.

Sorting Algorithms

- There are many sorting algorithms, such as:
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort

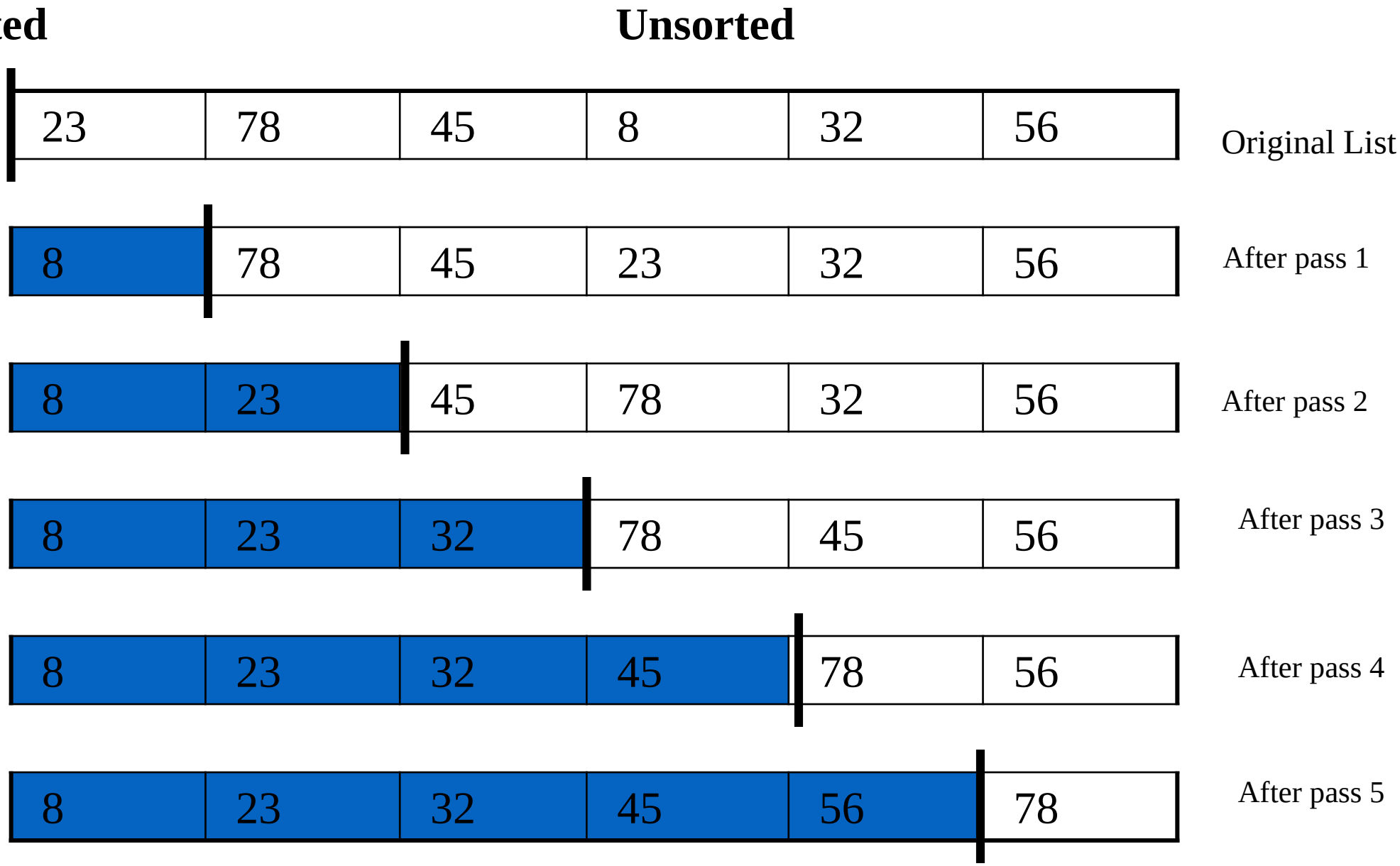
In-place vs out-of-place algorithms

- **Out of place sorting** is when *we need extra space for sorting purpose.* Such as **merge sort**

- **In-place sorting** algorithm directly modifies the list that it receives as input instead of creating a new list. Such as in **bubble sort, insertion sort, and selection sort**

Selection Sort

- Find the smallest element. Swap it with the first element.
- Find the second-smallest element. Swap it with the second element.
- This algorithm is called selection sort because it repeatedly selects the next-smallest element and swaps it into place.


Sorted
Unsorted
23 78 45 8 32 56
Original List
8 78 45 23 32 56
After pass 1
8 23 45 78 32 56
After pass 2
8 23 32 78 45 56
After pass 3
8 23 32 45 78 56
After pass 4
8 23 32 45 56 78
After pass 5

Selection Sort (code)

```
void selectionSort( Item a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
       if (a[j] < a[min]) min = j;

    swap(a[i], a[min]);
  }
}
```

```
void swap( Object &lhs, Object &rhs )
{
  Object tmp = lhs;
  lhs = rhs;
  rhs = tmp;
}
```

Selection Sort (Code JAVA)

```
public static void selectionSort(int[] arr){
        for (int i = 0; i < arr.length - 1; i++)
        {
            int index = i;
            //Finding the lowest index
            for (int j = i + 1; j < arr.length; j++){
                if (arr[j] < arr[index]){
                    index = j;
                }
            }
            //swap
            int smallerNumber = arr[index];
            arr[index] = arr[i];
            arr[i] = smallerNumber;
        }
    }
```

Selection Sort (Advantages)

- Easy to implement

- If we need to sort the first three elements only, what we do? Compare with other sorting algorithms.

- mostly used with small lists (Arrays)

- No need for extra memory (considered **In-place sorting** algorithm)

Selection Sort (Disadvantages)

- Slow algorithm:

"its time complexity $O(n^2)$"

- Not Smart Algorithm:

(blind algorithm) (Compare it with Bubble sort)

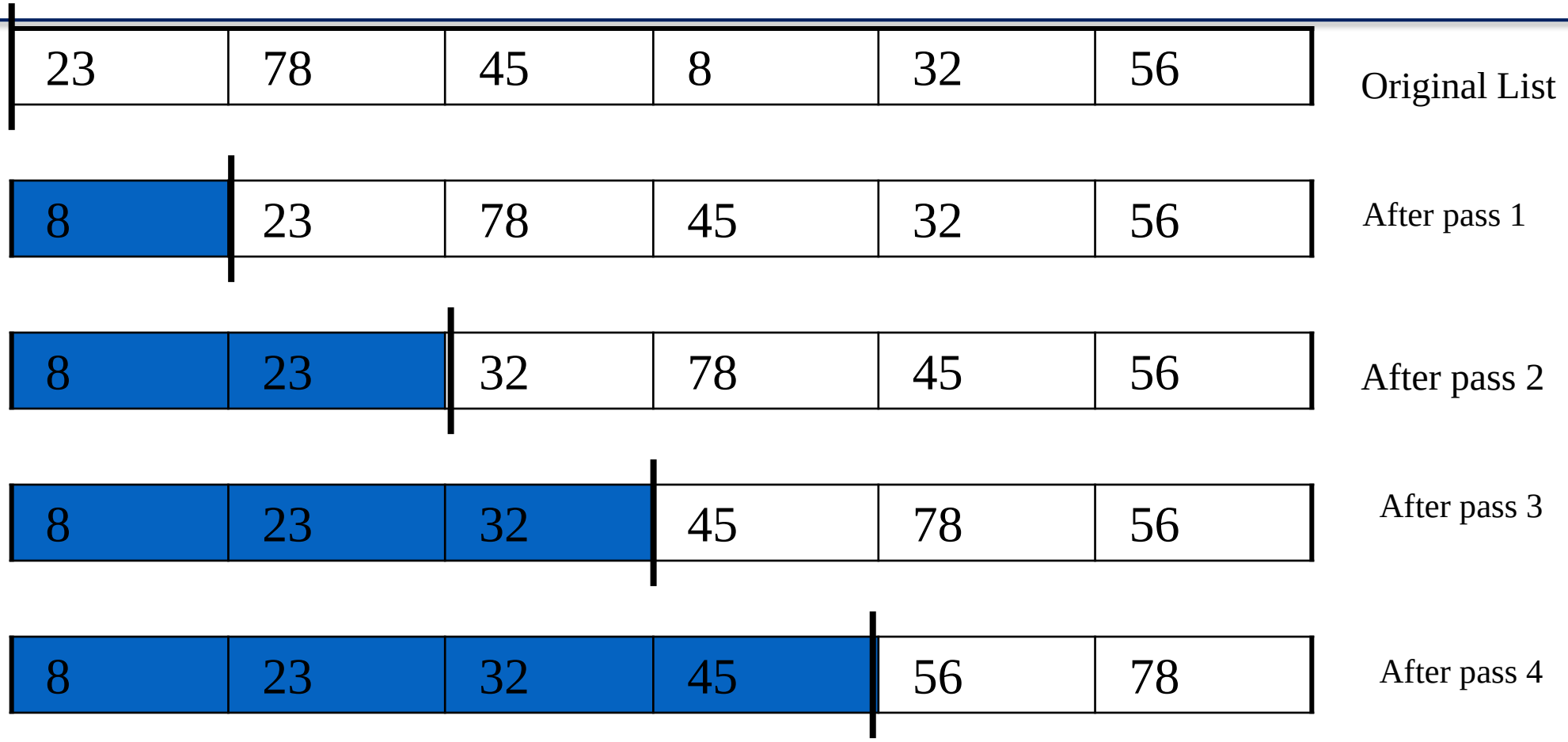
If the input list is sorted already, this algorithm do the same operations

best case time complexity is $O(n^2)$

Bubble Sort

- The list is divided into two sub-lists: sorted and unsorted.
- The smallest element is bubbled from the unsorted list and moved to the sorted sub-list.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

Bubble Sort (Ascending order)



Bubble Sort Algorithm

```
template <class Item>
void bubleSort(Item a[], int n)
{
   bool sorted = false;
   int last = n-1;

   for (int i = 0; (i < last) && !sorted; i++){
      sorted = true;
      for (int j=last; j > i; j--)
         if (a[j-1] > a[j]{
            swap(a[j],a[j-1]);
            sorted = false; // signal exchange
         }
    }
}
```

Bubble Sort – Analysis

- ***Best-case:*** **➔ O(n)**
  - Array is already sorted in ascending order.
  - The number of moves: 0 ➔ O(1)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** **➔ $O(n^2)$**
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$

Bubble Sort (Advantages)

- Smart Algorithm

Best case time complexity is O(n)

- Easy to implement
- mostly used with small lists (Arrays)
- No need for extra memory (considered **In-place sorting** algorithm)
- If we need to sort the first three elements only, what we do? Compare with other sorting algorithms.

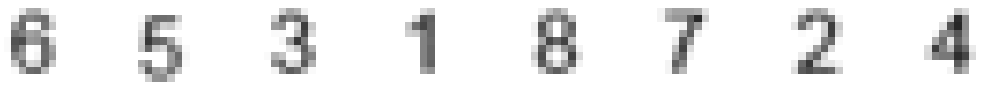
Bubble Sort (Disadvantages)

- Slow algorithm:

"its time complexity $O(n^2)$"

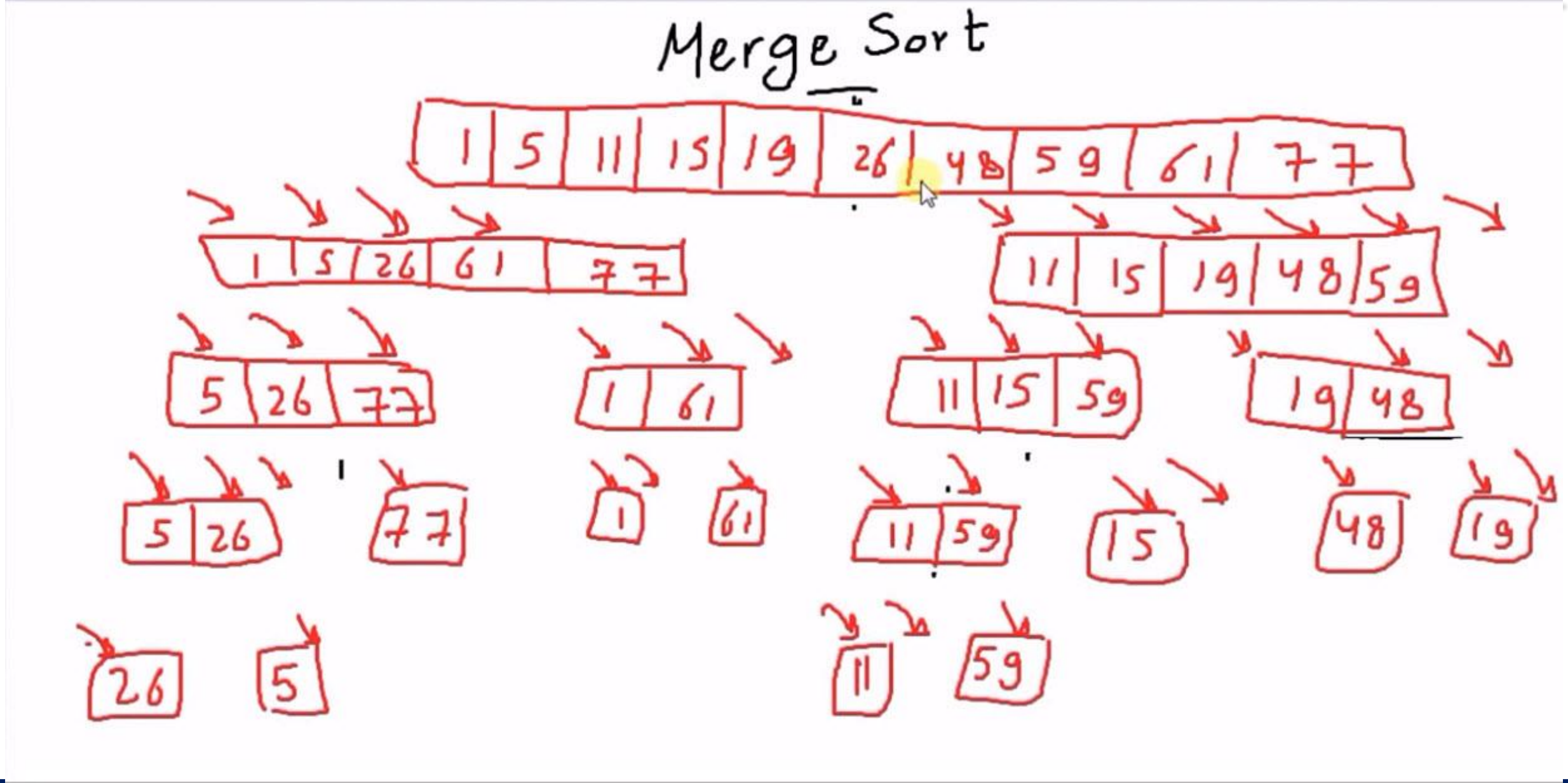
MERGE SORT

Merge Sort

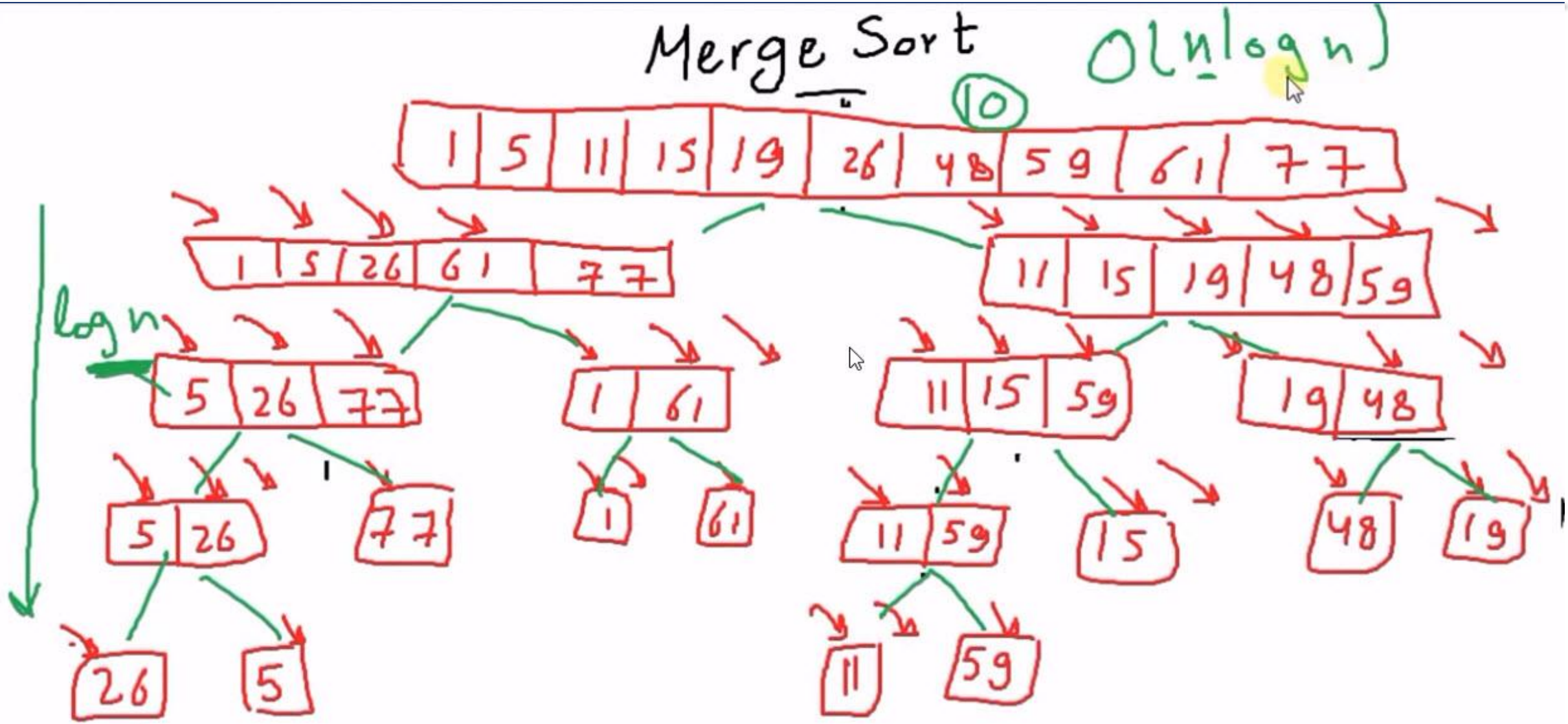


https://en.wikipedia.org/wiki/Merge_sort

Merge Sort (Time Complexity)



Merge Sort (Time Complexity)



Merge Sort (Advantages)

Fast algorithm

Its complexity O(n.log n) “base 2”

Is faster than other algorithms with complexity $O(n^2)$

Merge Sort (Disadvantages)

- Not Smart Algorithm (Compare it with Bubble sort)
- If the input list is sorted already, this algorithm do the same operations

- Extra memory to run this algorithm

To merge two list of size 1,000,000, we need a new empty list of size 2,000,000

Compare it with other algorithm that don't need extra memory

Merge Sort (recursive version)

```
def mergeSort(myList):
    if len(myList) > 1:
        mid = len(myList) // 2
        left = myList[:mid]
        right = myList[mid:]
        mergeSort(left)
        mergeSort(right)
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                myList[k] = left[i]
                i += 1
            else:
                myList[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            myList[k] = left[i]
            i += 1;       k += 1
        while j < len(right):
            myList[k]=right[j]
            j += 1            k += 1
```

```
myList = [54,26,93,17,77,31,44,55,20]
mergeSort(myList)
print(myList)
```

The steps:

1. The list is divided into left and right in each recursive call until two adjacent elements are obtained.

2. Now begins the sorting process. The i and j iterators traverse the two halves in each call. The k iterator traverses the whole lists and makes changes along the way.

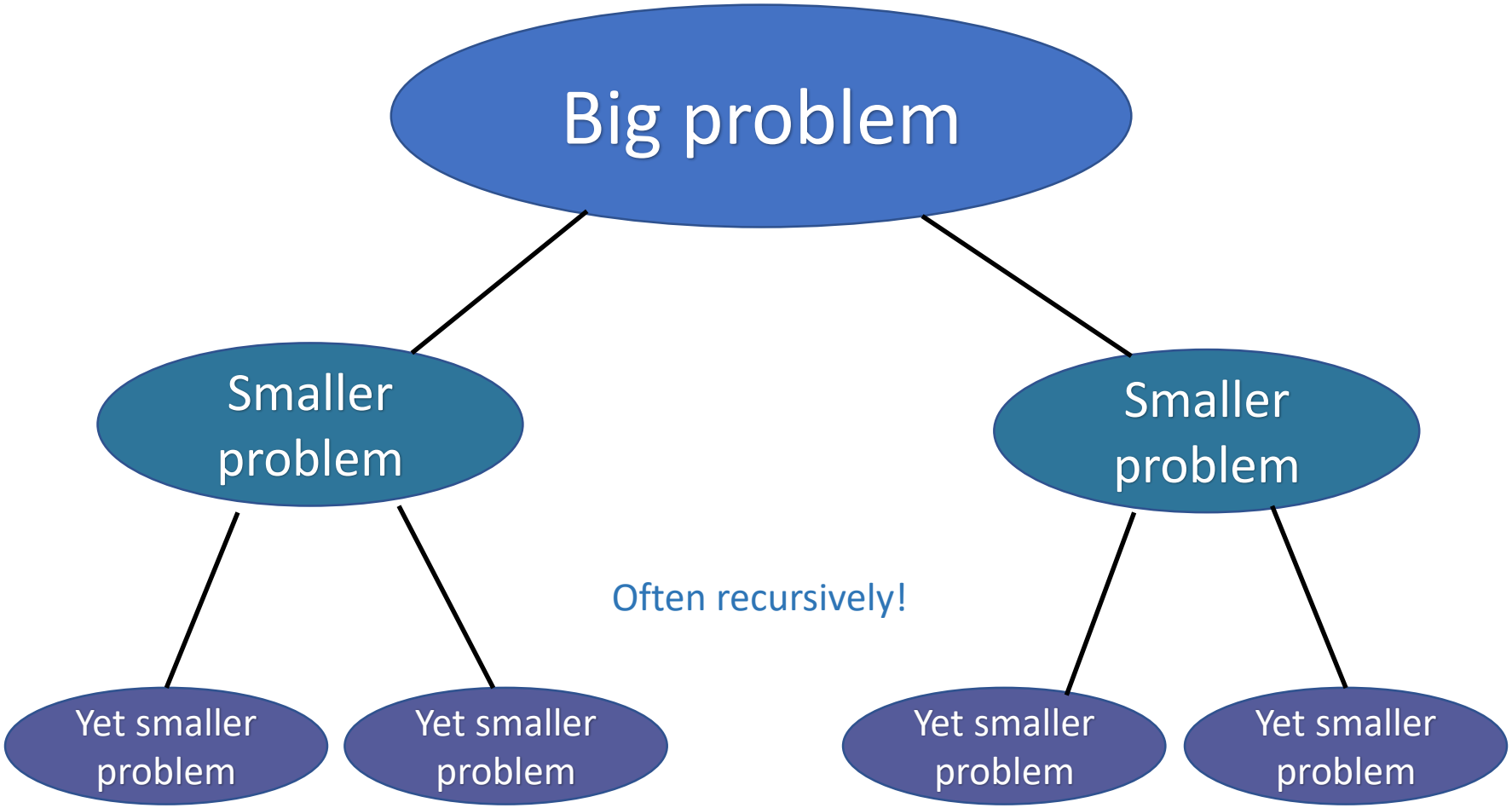
3. If the value at i is smaller than the value at j, left[i] is assigned to the myList[k] slot and i is incremented. If not, then right[j] is chosen.

4. This way, the values being assigned through k are all sorted.

5. At the end of this loop, one of the halves may not have been traversed completely. Its values are simply assigned to the remaining slots in the list.

Divide and conquer

- Break problem up into smaller (easier) sub-problems



http://web.stanford.edu/class/archive/cs/cs161/cs161.1182/

Divide-and-Conquer

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 the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
3. Combine the solutions to the subproblems into the solution for the original problem.

Divide-and-Conquer

- When the subproblems are large enough to solve recursively, we call that the recursive case. Once the subproblems become small enough that we no longer recurse,
- we say that the recursion “bottoms out” and that we have gotten down to the base case. Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem.
- We consider solving such subproblems as part of the combine step.

Merge sort Divide-and-Conquer

- The merge sort algorithm follows the divide-and-conquer paradigm.

1. Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements.
2. Conquer: Sort the two subsequences recursively using merge sort.
3. Combine: Merge the two sorted subsequences to produce the sorted answer.

- The recursion “bottoms out” when the sequence to be sorted has length 1.Every sequence of length 1 is already in sorted order

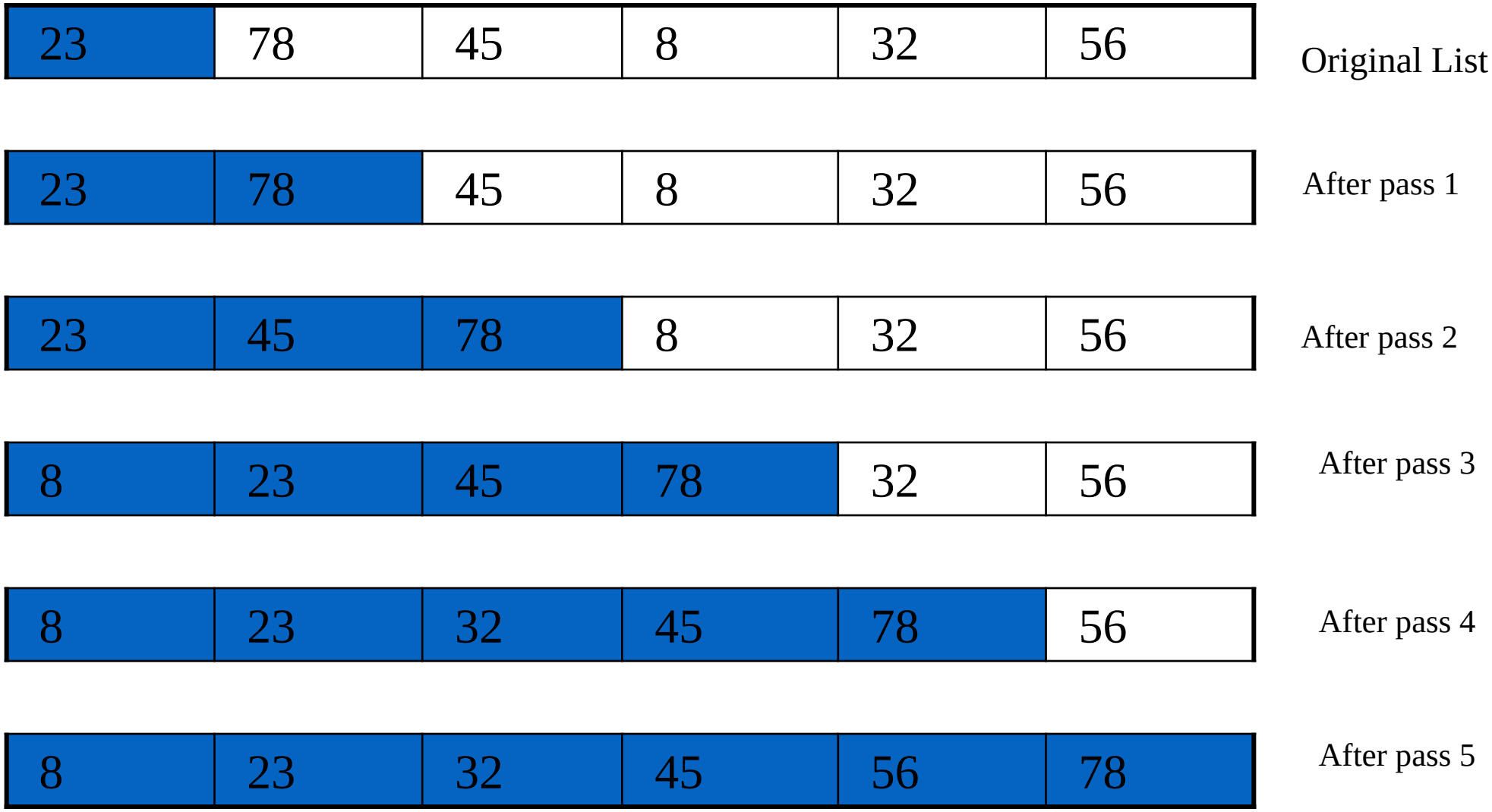
Merge sort dance

- https://www.youtube.com/watch?v=XaqR3G_NVoo

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sub-list, and inserted at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

Sorted **Unsorted**



Insertion Sort Algorithm

```
template <class Item>
void insertionSort(Item a[], int n)
{
   for (int i = 1; i < n; i++)
   {
      Item tmp = a[i];

      for (int j=i; j>0 && tmp < a[j-1]; j--)
         a[j] = a[j-1];
      a[j] = tmp;
   }
}
```

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 2,3, …, n
  - The number of moves: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- ***Average-case:*** ➔ **$O(n^2)$**
- **So, Insertion Sort is $O(n^2)$**

Quicksort

- Like mergesort, Quicksort is also based on

  the *divide-and-conquer* paradigm.
- But it uses this technique in a somewhat opposite manner,

  as all the hard work is done *before* the recursive calls.
- It works as follows:

  1. First, it partitions an array into two parts,
  2. Then, it sorts the parts independently,
  3. Finally, it combines the sorted subsequences by

     a simple concatenation.

Quicksort (cont.)

The quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
   - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
   - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Recursion***: Recursively sort the sublists separately.
3. ***Conquer***: Put the sorted sublists together.