Fast Sorting

Lecture 18

Menu

- Sorting
  - Design by Divide and Conquer
  - Merge Sort
  - QuickSort

Slow Sorts

- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - O(n^2)

- Problem:
  - Insertion and Bubble
    - only compare adjacent items
    - only move items one step at a time
  - Selection
    - compares every pair of items –
      - ignores results of previous comparisons.

- Solution:
  - compare and swap items at a distance
  - do not perform redundant comparisons

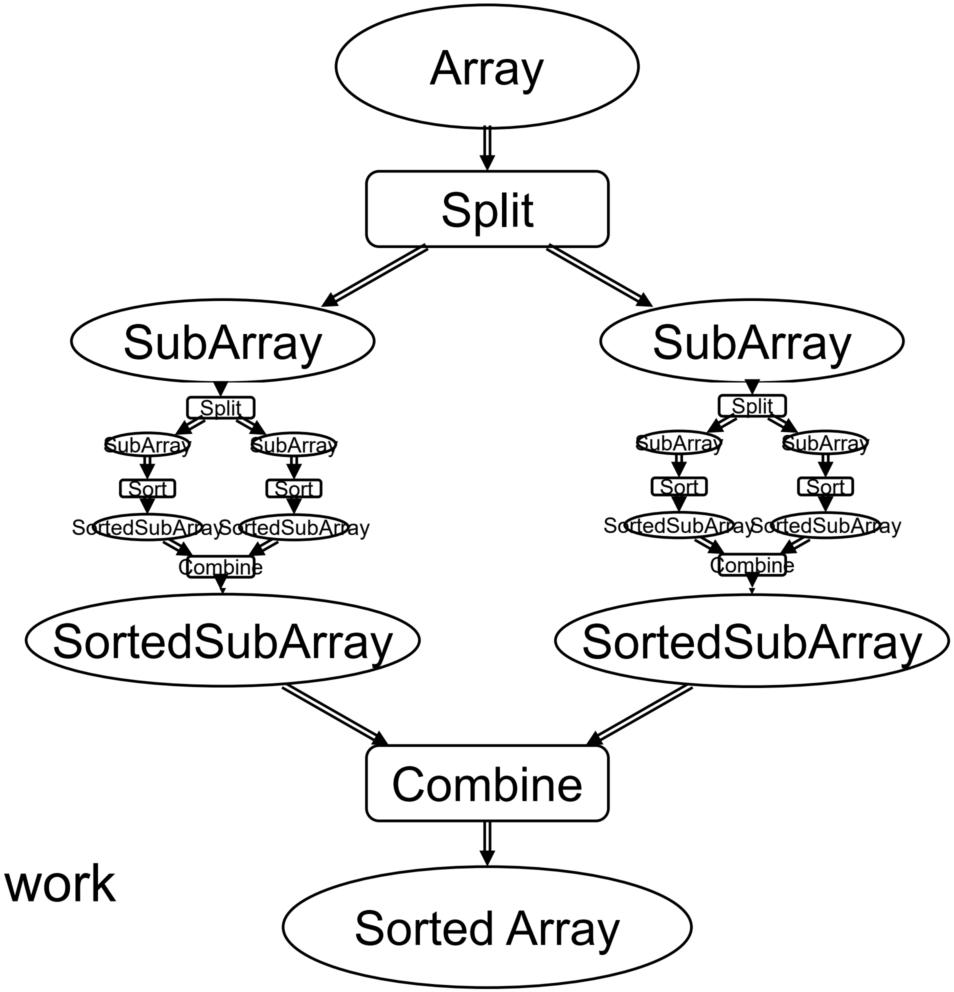
Divide and Conquer Sorts

To Sort:

- Split
- Sort each part (recursive)
- Combine

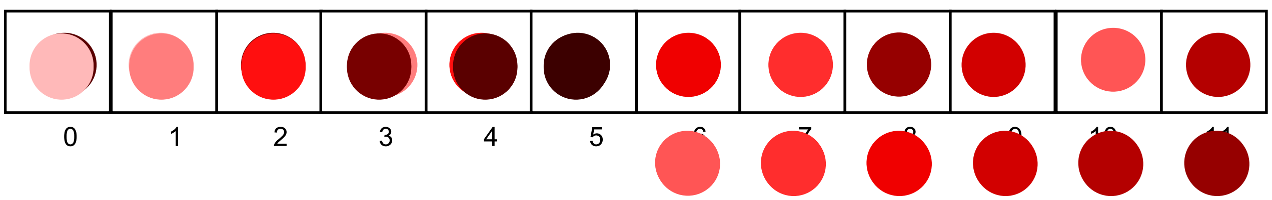
Where does the work happen?

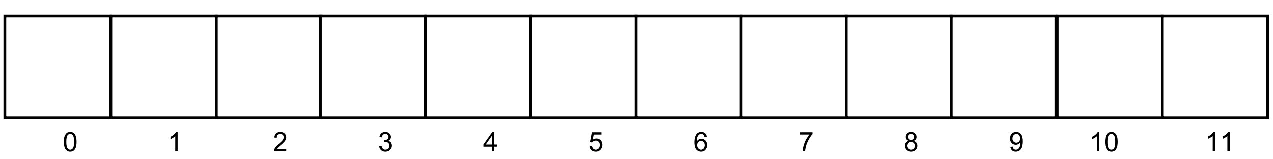
- MergeSort:
  - split trivial
  - combine does all the work
- QuickSort:
  - split does all the work
  - combine trivial



Merge Sort

- Split the array exactly in half
- Sort each half
- “Merge” them together.

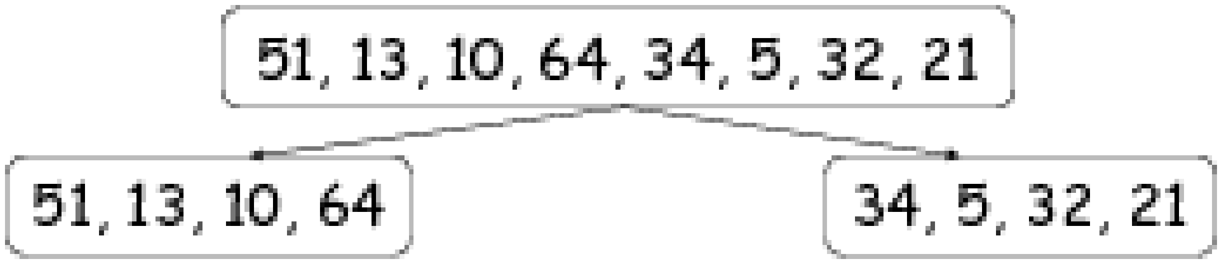




Need a temporary array

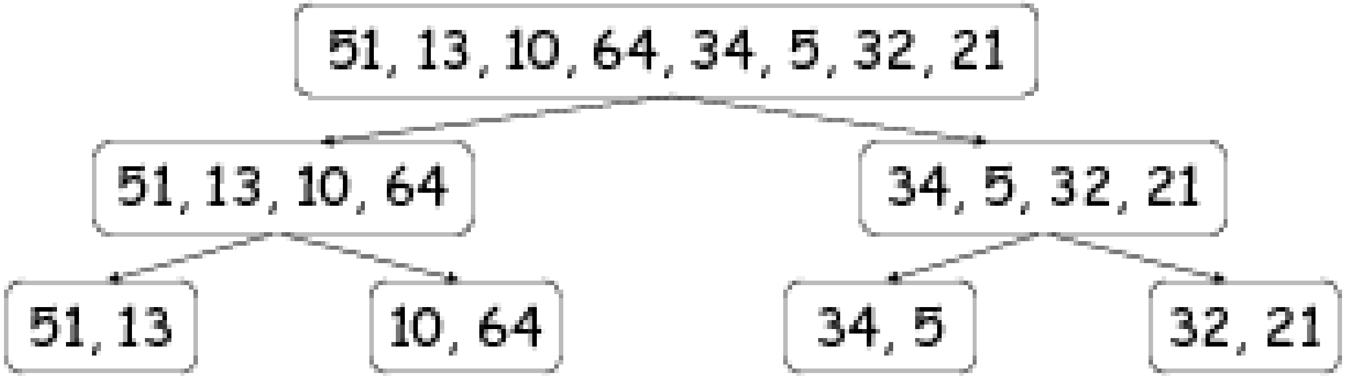
51, 13, 10, 64, 34, 5, 32, 21

we want to sort these 8 numbers,
divide them into two halves

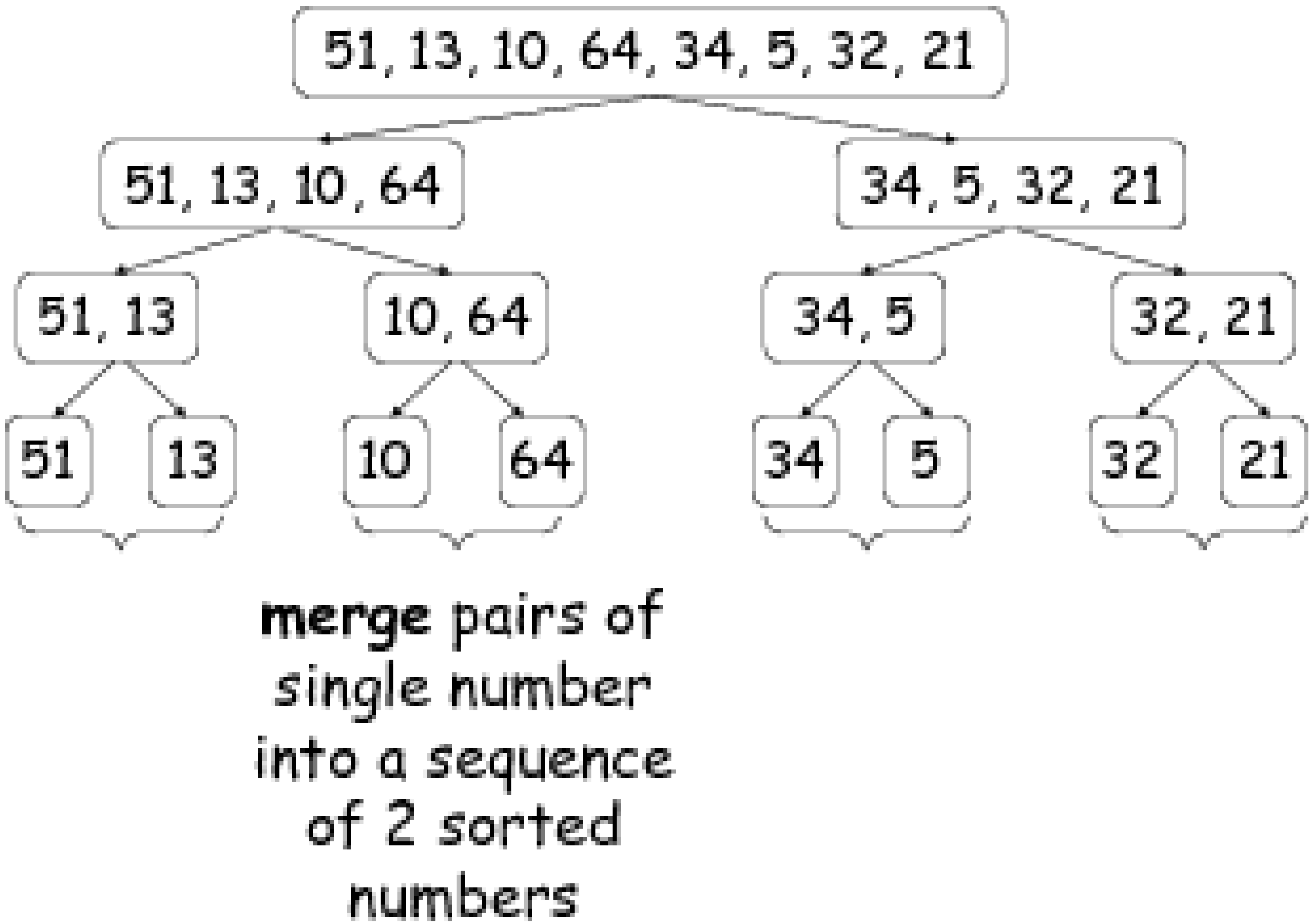


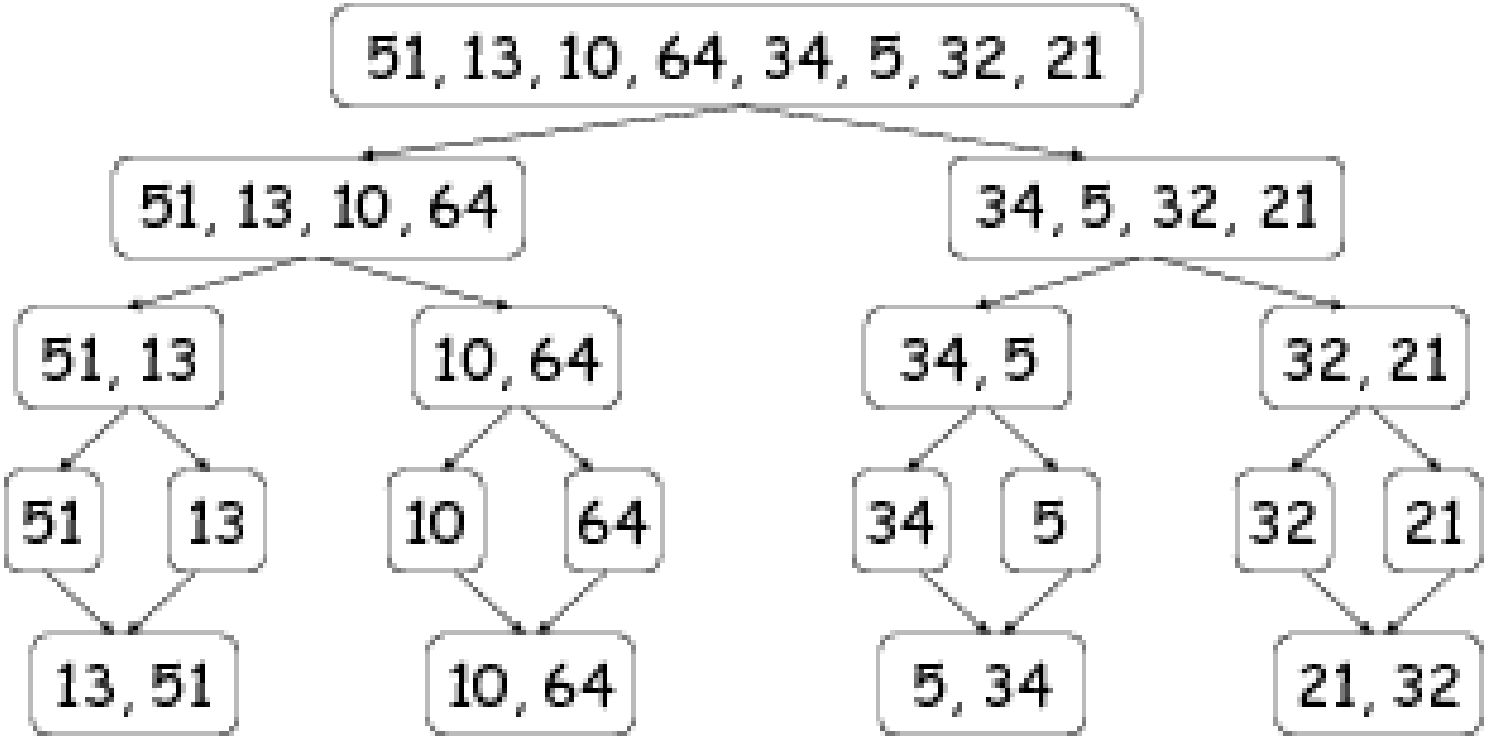
divide these 4 numbers into halves

similarly for these 4

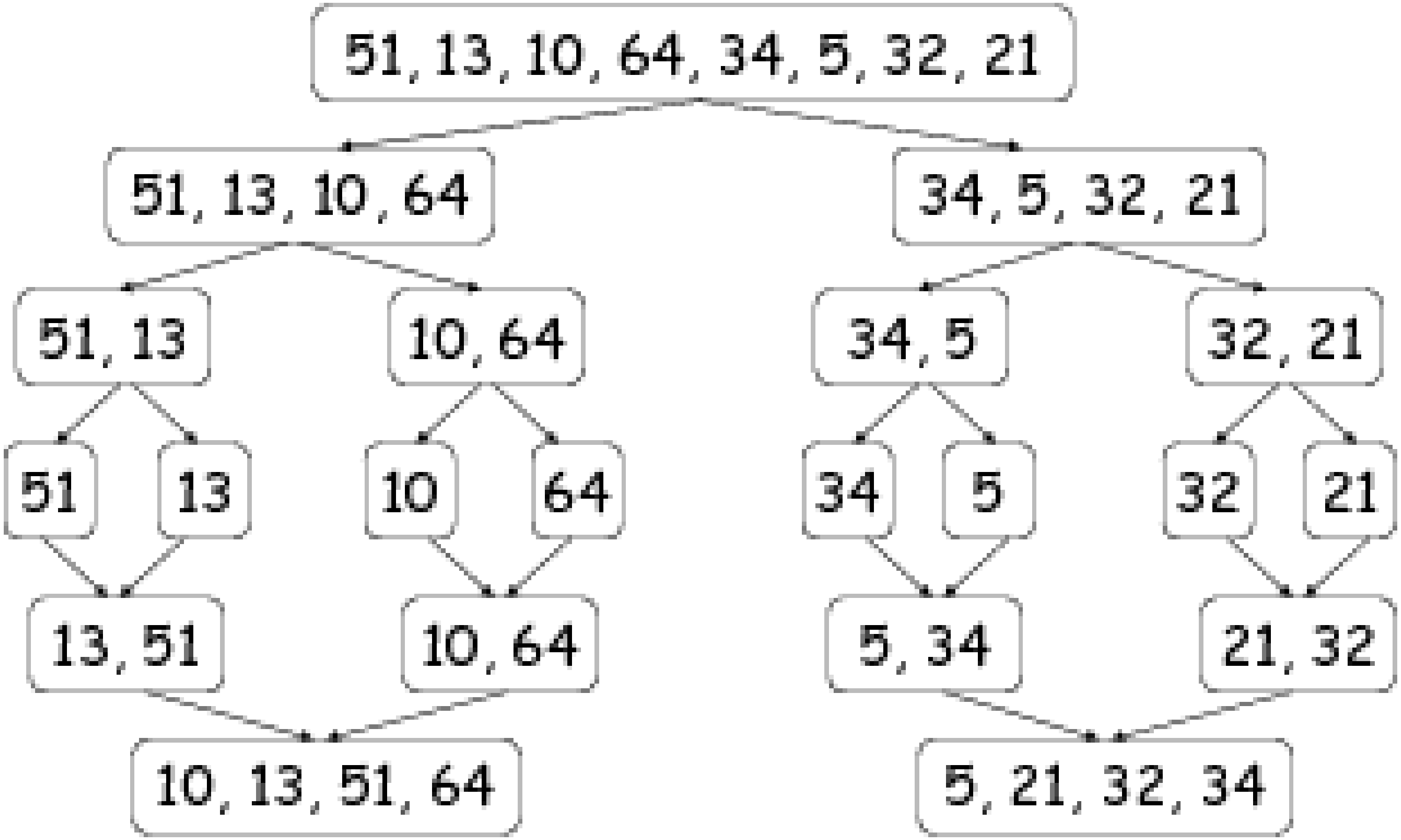


further divide each shorter sequence ...
until we get sequence with only 1 number

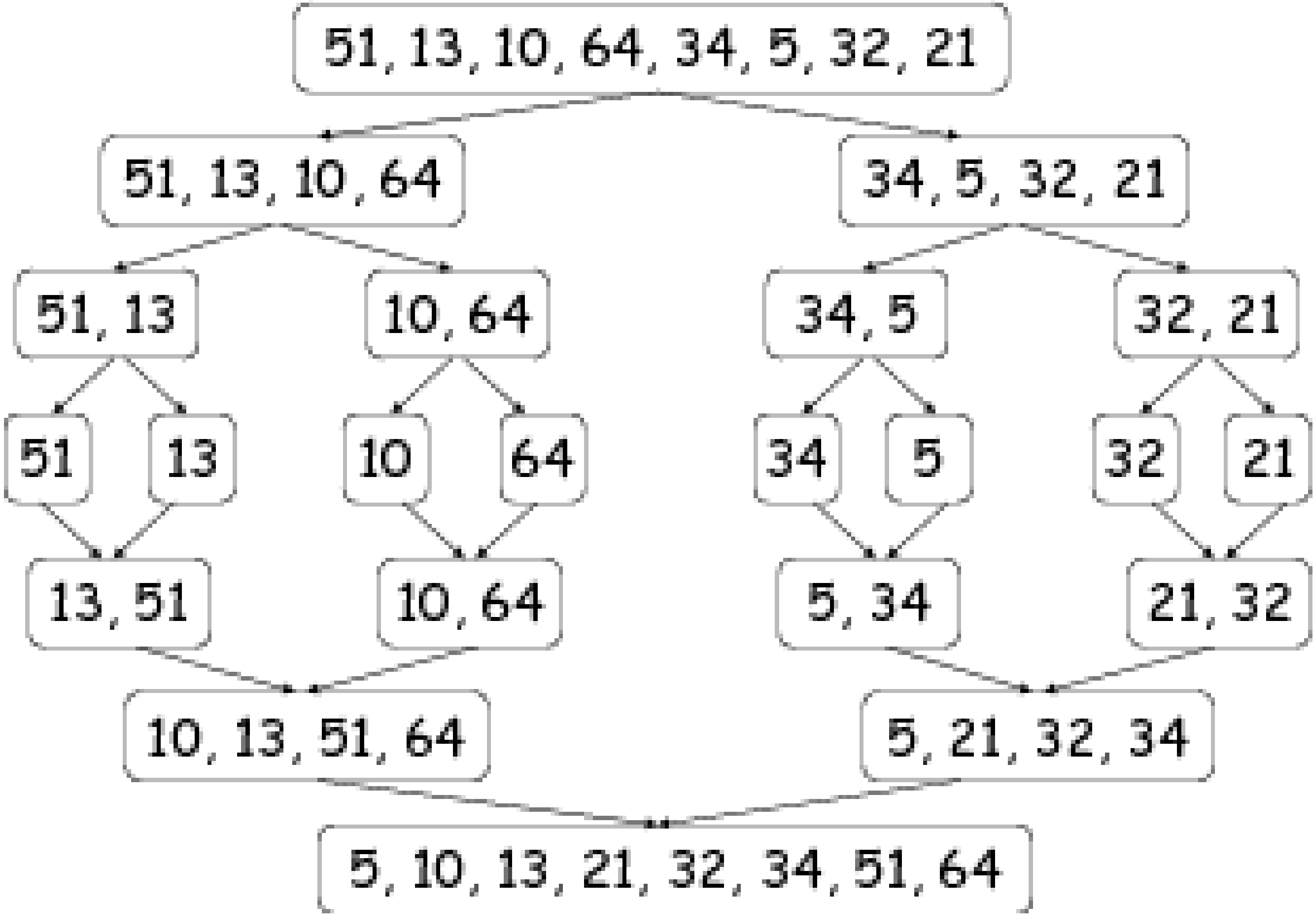

51, 13, 10, 64, 34, 5, 32, 21
51, 13, 10, 64
34, 5, 32, 21
51, 13
10, 64
34, 5
32, 21
51
13
10
64
34
5
32
21
merge pairs of single number into a sequence of 2 sorted numbers



then **merge** again into sequences of 4 sorted numbers



one more merge give the **final** sorted sequence


51, 13, 10, 64, 34, 5, 32, 21
51, 13, 10, 64
34, 5, 32, 21
51, 13
10, 64
34, 5
32, 21
51
13
10
64
34
5
32
21
13, 51
10, 64
5, 34
21, 32
10, 13, 51, 64
5, 21, 32, 34
5, 10, 13, 21, 32, 34, 51, 64

Mergesort – merging details

- given two sorted arrays, merge them into one sorted array
- keep track of the smallest element in each array, output the smaller of the two to a third array
- Continue until both arrays are exhausted
- If any array is exhausted first, then simply output the rest of another array

- This so-called 2-way merging can be generalized to multi-way merging

Merging process in details

3 4 7 33 78
2 11 54 69 71 82 99

find the smallest of each array; compare

← **2**
3 4 7 33 78
11 54 69 71 82 99

output the smaller **2**; remove 2 from the input array; find the smallest of each array; compare

← **2 3**
4 7 33 78
11 54 69 71 82 99

output the smaller **3**;remove 3 from the input array; find the smallest of each array; compare

← **2 3 4**
7 33 78
11 54 69 71 82 99

output the smaller **4**;remove 4 from the input array; find the smallest of each array; compare

← **2 3 4 7**
33 78
11 54 69 71 82 99

output the smaller **7**;remove 7 from the input array; find the smallest of each array; compare

← **2 3 4 7** 11

MergeSort

- Needs a temporary array for copying
  - create a temporary array
  - [fill with a copy of the original data.]

```
public static <E> void mergeSort(E[] data, int size,
                                  Comparator<E> comp){
    E[] other = (E[])new Object[size];
    for (int i=0; i<size; i++) other[i]=data[i];
    mergeSort(data, other, 0, size, comp);
}
```

MergeSort


```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
    // sort items from low..high-1 using temp array
    if (high > low+1){
        int mid = (low+high)/2;
        // mid = low of upper 1/2, = high of lower half.
        mergeSort(data, temp, low, mid, comp);
        mergeSort(data, temp, mid, high, comp);
        merge(data, temp, low, mid, high, comp);
        for (int i=low; i<high; i++)   data[i]=temp[i];
    }
}
```

Sort each half

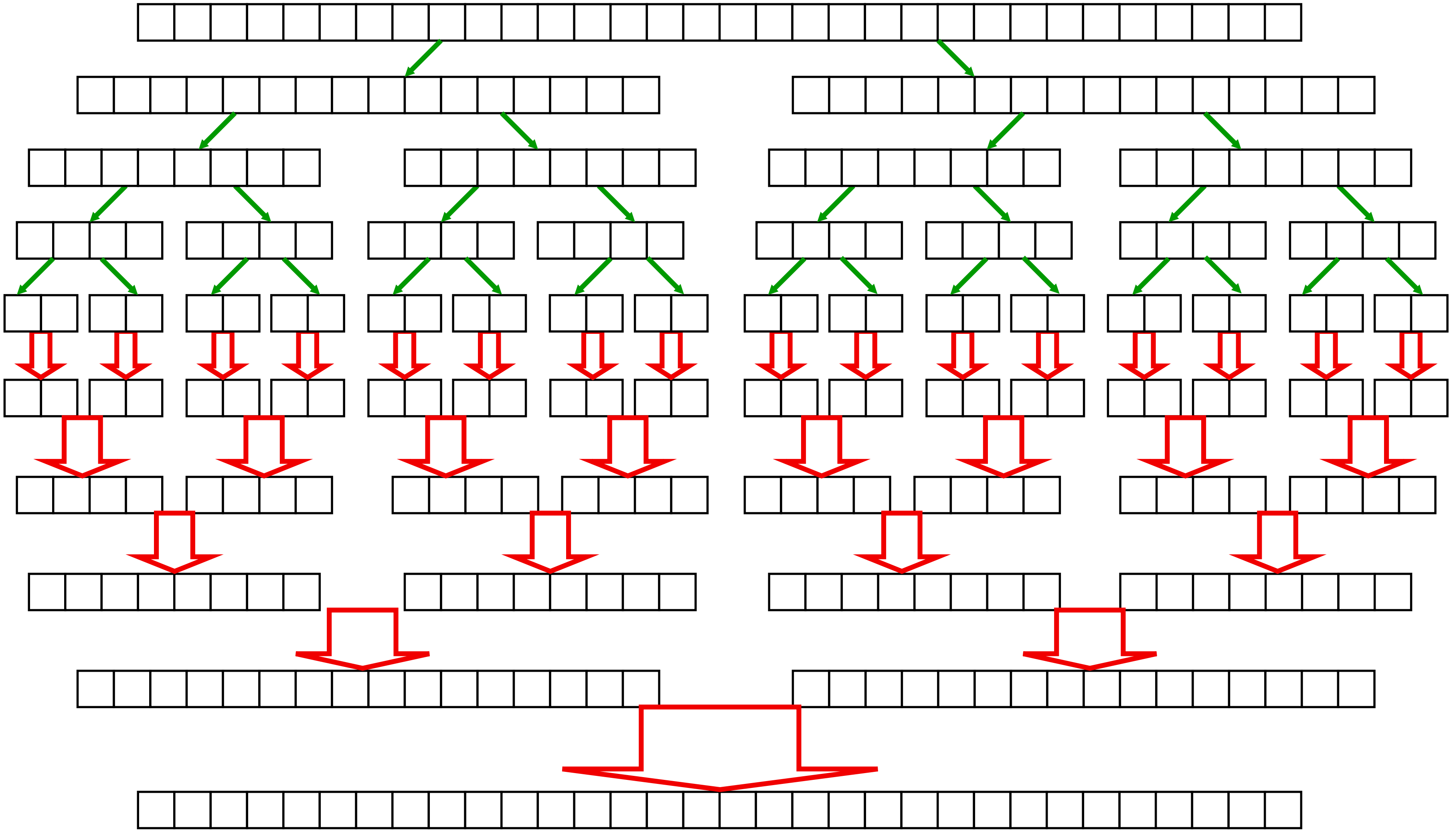
merge into temp

copy back

Merge

```
/** Merge from[low..mid-1] with from[mid..high-1] into  to[low..high-1.*/
private static <E> void merge(E[] from, E[] to, int low, int mid, int high,
                                                Comparator<E> comp){
    int index = low;          // where we will put the item into "to“
    int indxLeft = low;       // index into the lower half of the "from" range
    int indxRight = mid;      // index into the upper half of the "from" range
    while (indxLeft<mid && indxRight < high){
        if (comp.compare(from[indxLeft], from[indxRight]) <=0)
            to[index++] = from[indxLeft++];
        else
            to[index++] = from[indxRight++];
    }
    //copy over the remainder. Note only one loop will do anything.
    while (indxLeft<mid)
        to[index++] = from[indxLeft++];
    while (indxRight<high)
        to[index++] = from[indxRight++];
}
```

MergeSort



Exercise

- Rewrite **mergeSort** to improve its performance

Quicksort

- Invented by C.A.R. Hoare
- Used more widely than others

- works well for different types of data
- divide & conquer on sorting

- **O(N log N) on average**
- $O(N^2)$ worst case

Goal of splitting

<p	**p**	>p

p: pivot

Quicksort ideas

- partition the array into two parts
- partitioning involves the selection of a[i] where the following conditions are met:
  - a[i] is in its final place in the array for some i
  - none in a[1], ... , a[i-1] is greater than a[i]
  - none in a[i+1], ... , a[r] is less than a[i]

- apply quicksort recursively to each part independently

Quicksort in process

7 4 3 9 0 8 6	find an ‘i’;use v = ‘6’ to compare
l **r** **v**	use two pointers **l** & **r**, **l** scan from the left, stop when a[l] > v; r scan from right, stop when a[r]<v
0 4 3 9 **7** 8 6	swap a[l] & a[r]
0 4 3 **9** **7** 8 6	scan again from where we stop
l=r	stop again (when l >= r); i is found
0 4 3 **6** 7 8 **9**	swap a[l] with v; now every element to the left of 6 is less than 6, every element to the right of 6 is greater than 6
0 **4** **3** 6 **7** **8** 9	apply the same process to each partition
l **r** **v** **l** **r** **v**	
0 **4** 3 6 **7** 8 9	right partition sorted.
l=r	left partition stops scanning, new i found
0 **3** **4**	swap a[l] with v (3); left partition sorted
0 3 4 6 7 8 9	Done.

QuickSort – in brief

- Divide and Conquer,
  but does its work in the "**split**" step
- It splits the array into two (possibly unequal) parts:
  - choose a "**pivot**" item
  - make sure
    - all items < pivot are in the left part
    - all items > pivot are in the right part
- Then (**recursively**) sorts each part

```
public static <E> void quickSort(E[] data, int size, Comparator<E> comp){
    quickSort(data, 0, size, comp);
}
```

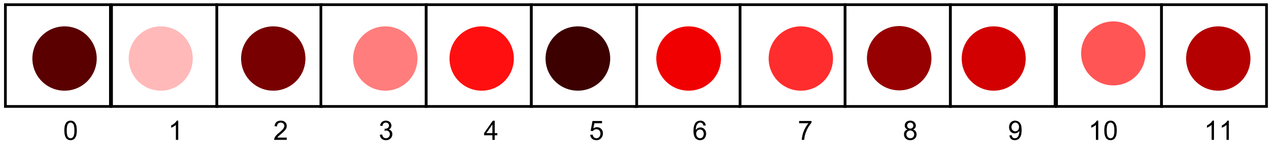
Quicksort in Python

- The following quickSort code in Python from Wikipedia.

```python
def quickSort(arr):
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more
```

QuickSort in Java

```
public static <E> void quickSort(E[] data, int low, int high,
                                 Comparator<E> comp){
    if (high-low < 2)      // only one item to sort.
        return;
    else {
        // split into two parts,  mid = index of boundary
        int mid = partition(data, low, high, comp);
        quickSort(data, low, mid, comp);
        quickSort(data, mid, high, comp);
    }
}
```



Reflection upon Quicksort

- Quicksort makes use of ONE pivot element for partitioning.
- What if more than one pivot is used for partitioning?
- Is it feasible?

- What are the advantages of multi-pivot quicksort if feasible?

Summary

- Sorting
  - Design by Divide and Conquer
  - Merge Sort
  - QuickSort

Readings

- [Mar07] Read 7.7, 7.8
- [Mar13] Read 7.6, 7.7