Merge-Sort

EECS 233

Previous Lecture: Quick-Sort

- Quick-Sort: a recursive, divide-and-conquer algorithm:
  - *divide:* partition the array into two subarrays so that :
    - *each element in the left array <= each element in the right array*
  - *conquer:* apply quick-sort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* nothing needs to be done, because of the criterion used in forming the subarrays

- Implementation of Quick-Sort
  - Choosing a good pivot value
  - Partitioning procedure
  - Recursive method

- Analysis of Quick-Sort running time
  - Best-case O(nlogn) and worst-case O(n^2)

An exercise

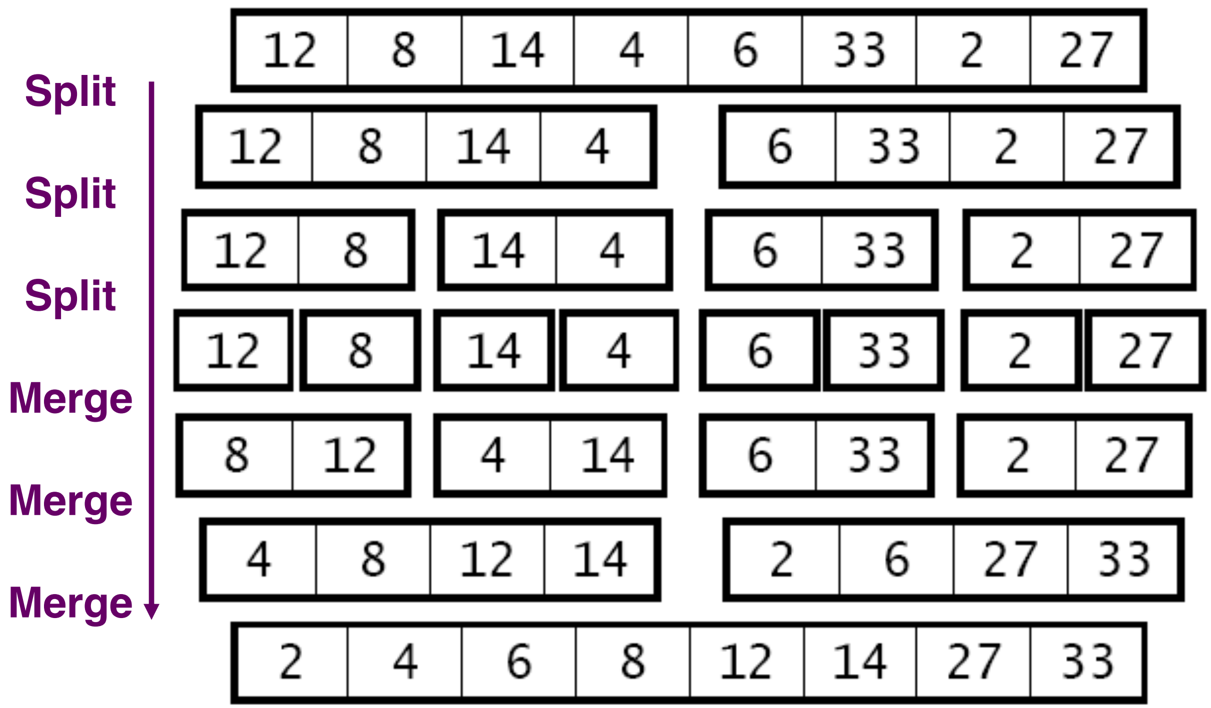
i

0	1	2	3	4	5	6	7	8
16	8	13	2	9	15	4	12	24

j

Merge-Sort

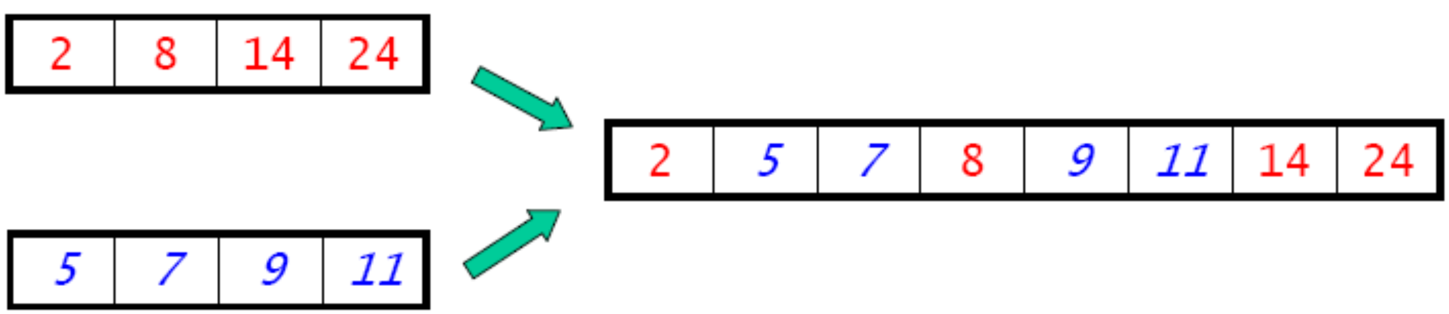
- Like quick-sort, merge-sort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply merge-sort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays



Merge-Sort

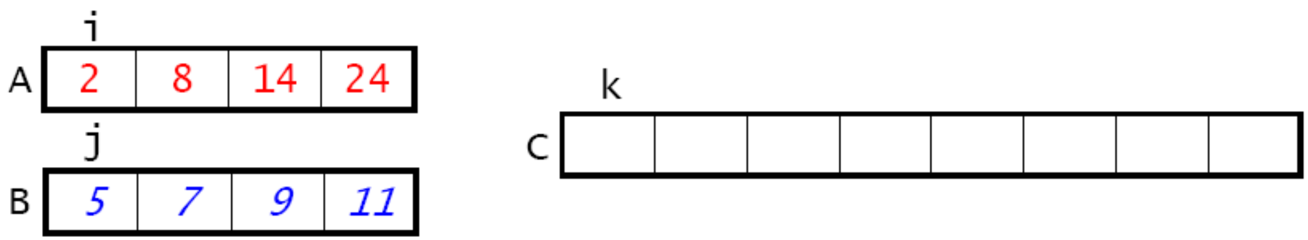
- All of the sorting algorithms we've seen thus far have sorted the array in place. They used only a small amount of additional memory, i.e., O(logn) additional space (for recursion)

- Merge-sort is a sorting algorithm that requires an additional temporary array of the same size as the original one.
  - it needs O(n) additional space, where n is the array size
  - space for *merging* two sorted arrays into a single sorted array.

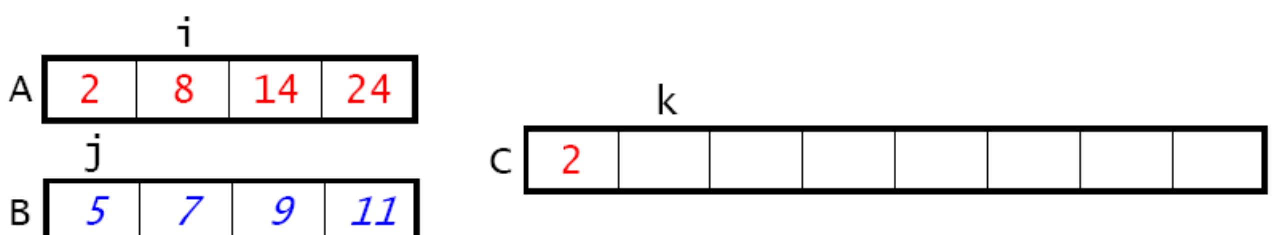


Merging Sorted Subarrays

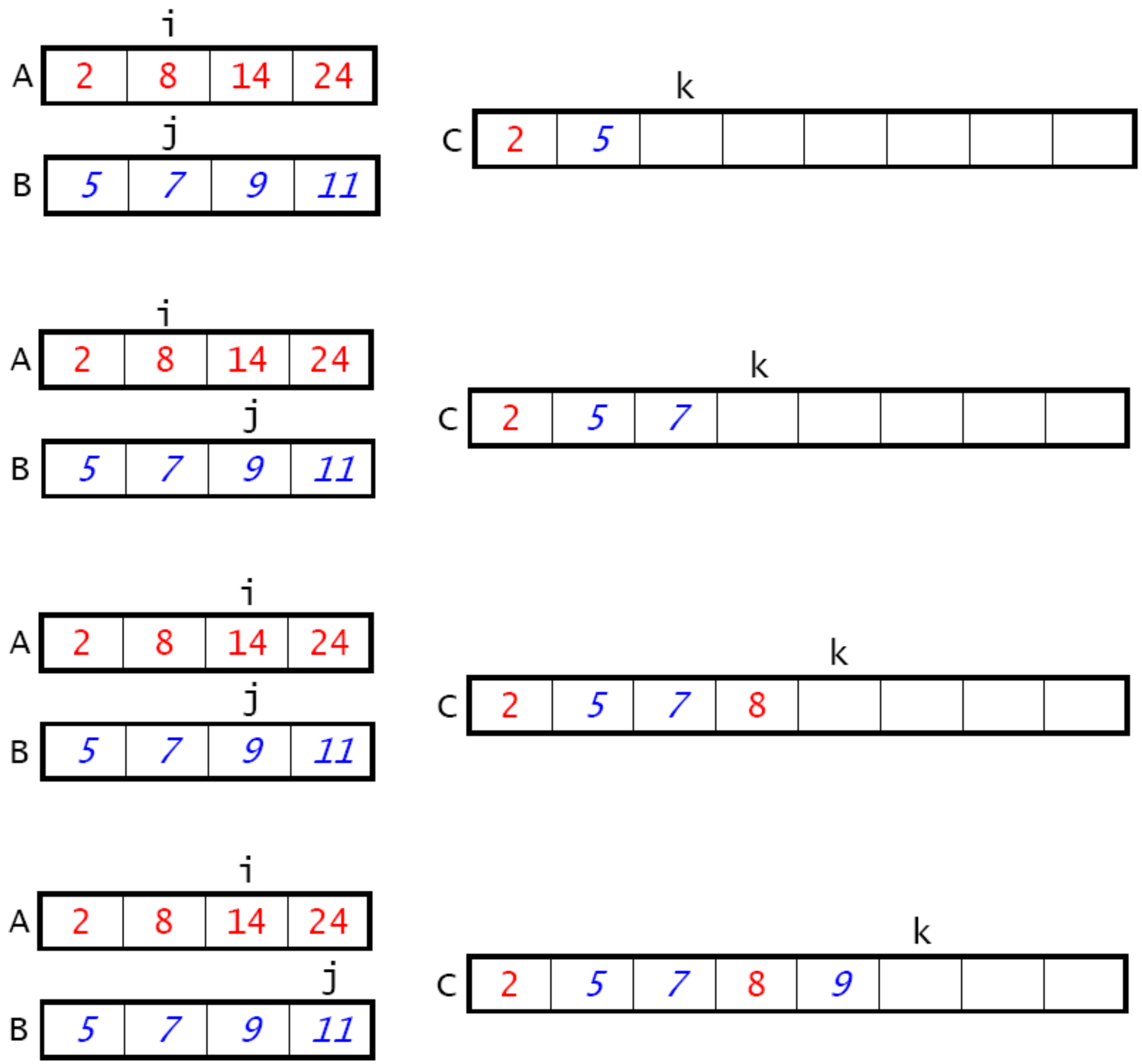
- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:



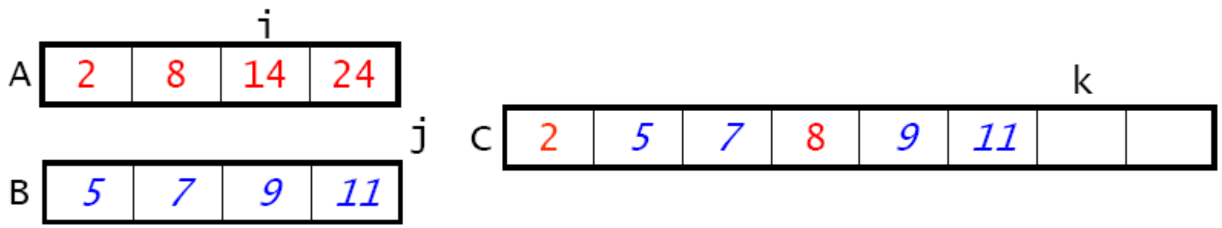
- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

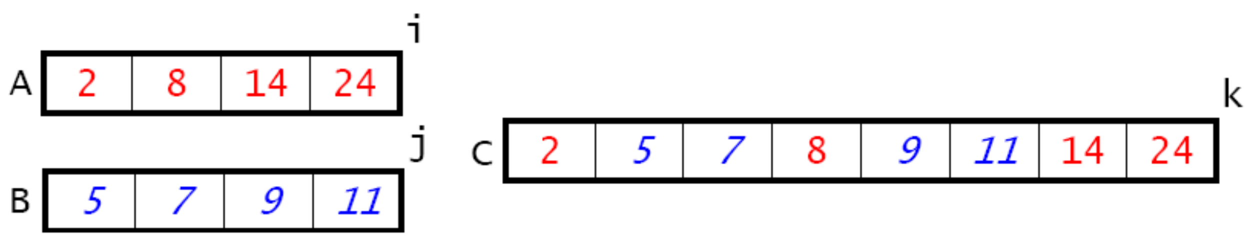


Merging Sorted Subarrays - Steps



Merging Sorted Subarrays - Steps





- Comparisons stop when either index reaches the end of its subarray
- The remaining elements in the other subarray are copied to the combined array C

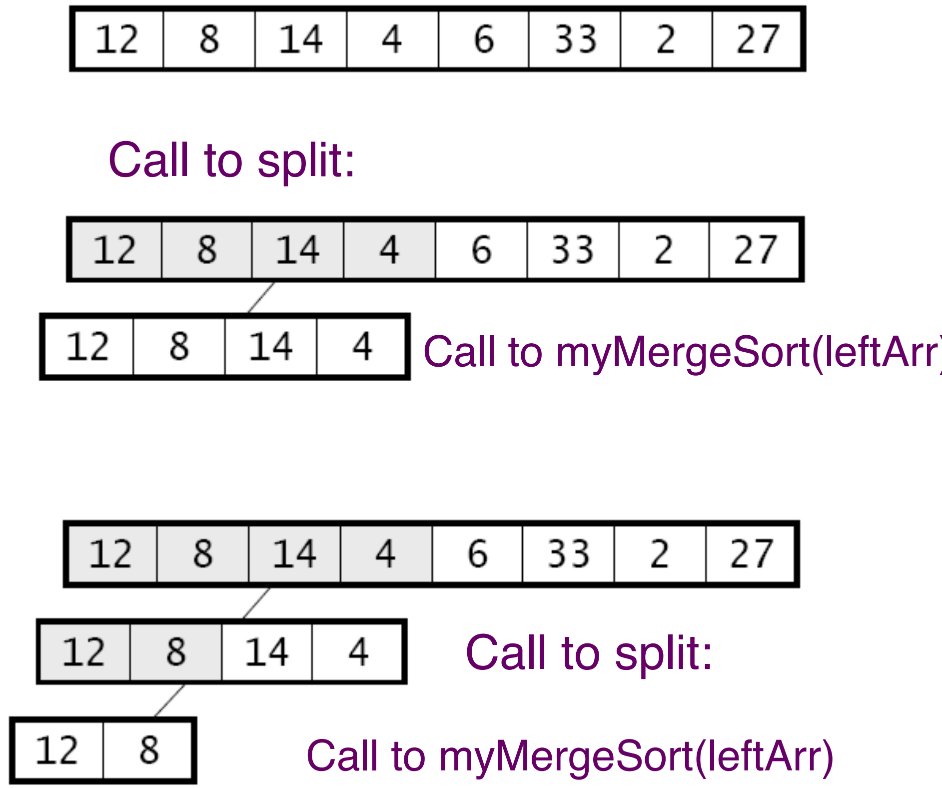
Recursive Procedure - Skeleton

- Assume we have the merge() method, we will write a recursive method to implement the divide-and-conquer approach.

```
static void mergeSort(int[] arr) {
        myMergeSort(arr);
}

static void myMergeSort(int[] arr) {
        if (arr.length == 1) return; // Base case
        // Allocate leftArr and rightArr
        split(arr,leftArr,rightArr);
        myMergeSort(leftArr);
        myMergeSort(rightArr);
        Merge(leftArr,RightArr,arr);
}
```

Recursive Calls - Steps

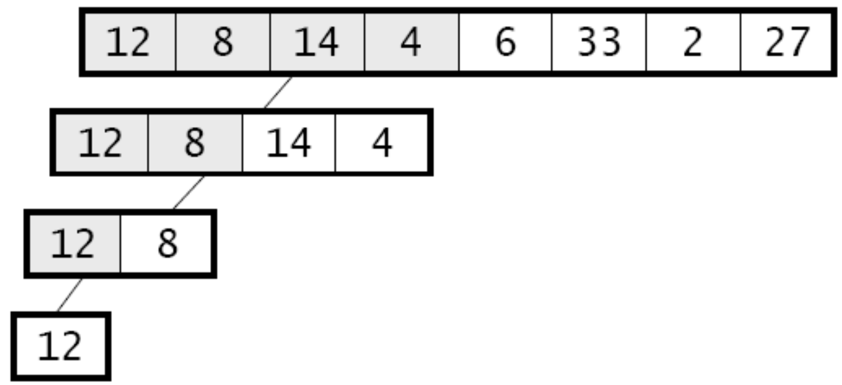
```
static void mergeSort(int[] arr) {
        myMergeSort(arr);
}

static void myMergeSort(int[] arr) {
        if (arr.length == 1) return;
        // Allocate leftArr and rightArr
        split(arr,leftArr,rightAr
        myMergeSort(leftArr);
        myMergeSort(rightArr);
        Merge(leftArr,rightArr,arr);

}
```



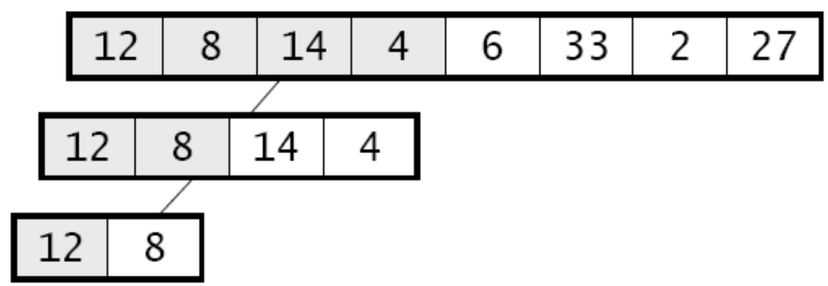
Recursive Calls - Steps

```
static void mergeSort(int[] arr) {
        myMergeSort(arr);
}
static void myMergeSort(int[] arr) {
        if (arr.length == 1) return;
        // Allocate leftArr and rightArr
        split(arr,leftArr,rightArr);
        myMergeSort(leftArr);
        myMergeSort(rightArr);
        Merge(leftArr,rightArr,arr);


}
```

Further split it into two size-1 subarrays, and issue recursive calls again



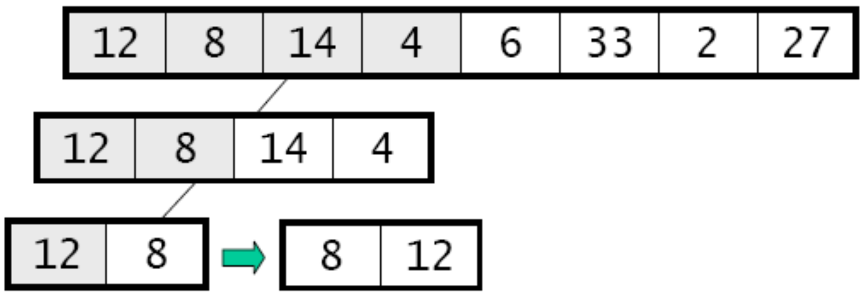
We are down to the base cases, so simply return (we have two sorted subarrays {12} and {8})



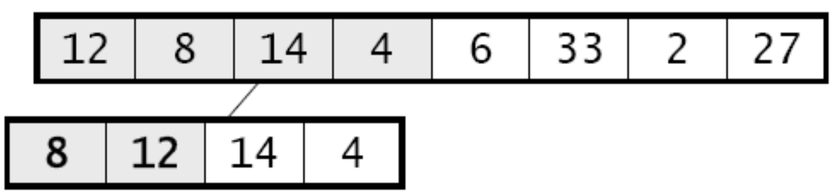
Recursive Calls - Steps

```
static void mergeSort(int[] arr) {
        myMergeSort(arr);
}

static void myMergeSort(int[] arr) {
        if (arr.length == 1) return;
        // Allocate leftArr and rightArr
        split(arr,leftArr,rightArr);
        myMergeSort(leftArr);
        myMergeSort(rightArr);
        Merge(leftArr,RightArr, arr);

}
```

Call merge() to merge two subarrays into original array

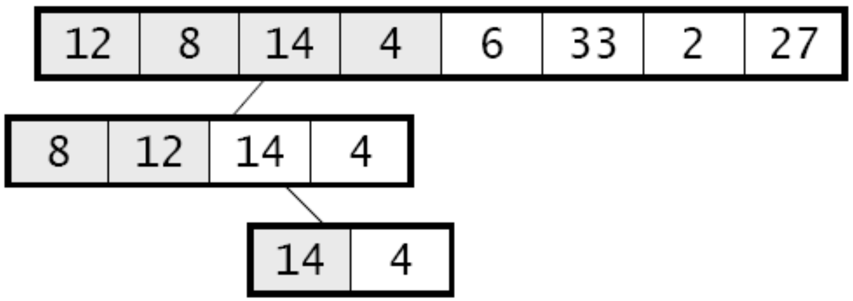


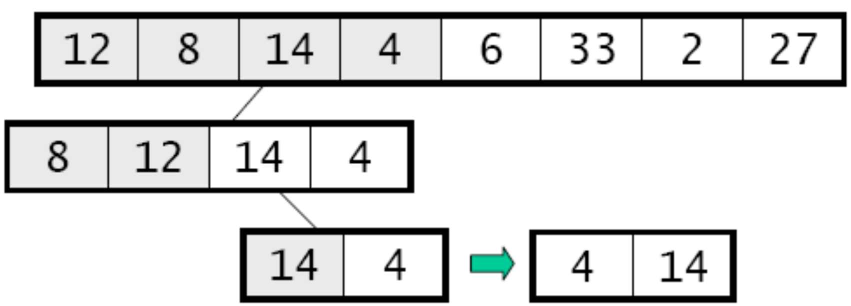
Return to the recursive call for the 4-element subarray, and start another recurise call for the right subarray {14, 4}.



Recursive Calls - Steps

Repeat the similar process for the right 2-element subarray

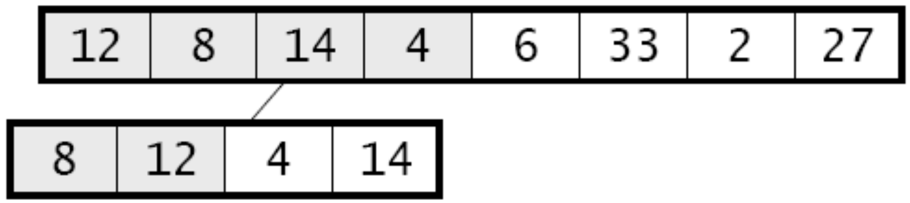




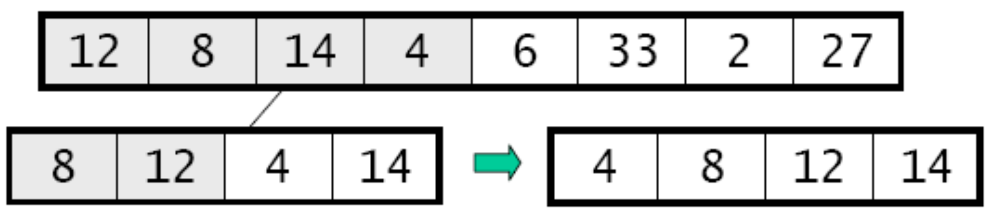

```
static void mergeSort(int[] arr) {
        myMergeSort(arr);
}

static void myMergeSort(int[] arr) {
         if (arr.length == 1) return;
        // Allocate leftArr and rightArr
        split(arr,leftArr,rightArr);
        myMergeSort(leftArr);
        myMergeSort(rightArr);
        Merge(leftArr,RightArr, arr);
}
```

Recursive Calls - Steps

Return from the recursive call for the 2-element right subarray. We have



Call merge() to merge the 2-element subarrays, and copy the elements back

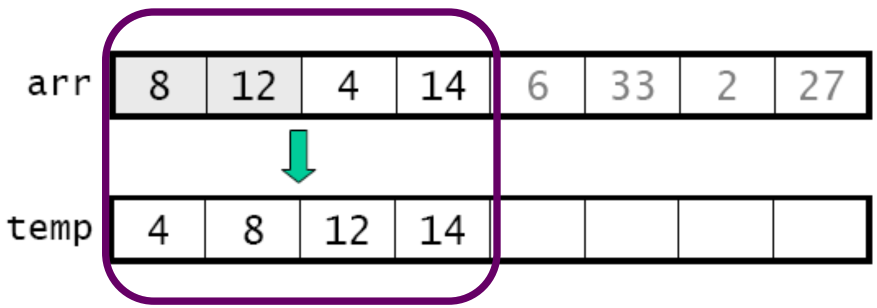


Etc.

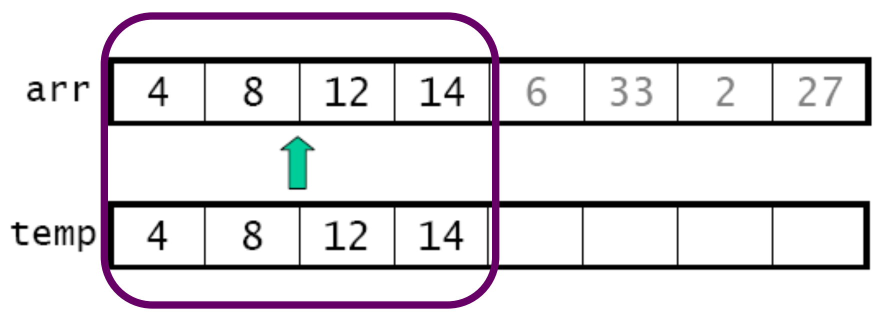
Implementation of Merge-Sort

- Our approach so far was to create new arrays for each new set of subarrays, and to merge them back into the array that was split.
  - Creates a lot of arrays in the recursive call chain

- Instead, we'll create a temp. array of the same size as the original.
  - pass it to each call of the recursive merge-sort method
  - use it when merging subarrays of the original array:

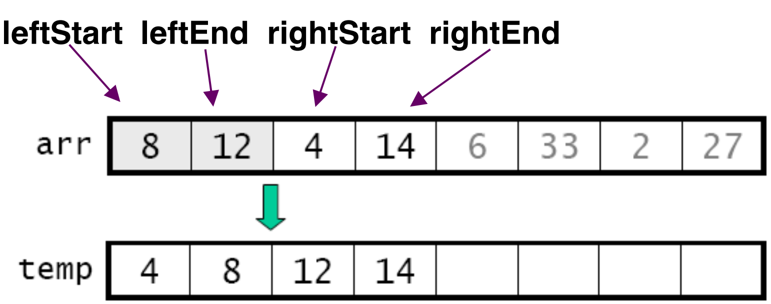


  - after each merge, copy the result back into the original array:



The Helper Method merge()

```
static void merge(int[] arr, int[] temp, int leftStart, int leftEnd, int rightStart, int rightEnd) {

    int i = leftStart; // index into left subarray
    int j = rightStart; // index into right subarray
    int k = leftStart; // index into temp
    while ( ? ) {


    }

     ?

    for (i = leftStart; i <= rightEnd; i++)     // copy back
        arr[i] = temp[i];
}
```



```
static void merge(int[] arr, int[] temp, int leftStart, int leftEnd, int rightStart, int rightEnd) {

        int i = leftStart; // index into left subarray
        int j = rightStart; // index into right subarray
        int k = leftStart; // index into temp
        while (i  <= leftEnd && j <= rightE) {

              if (array[i] <= array[j])
                        {
                           temp[k] = array[i];
                           i++;
                        }
                        else
                        {
                           temp[k] = array[j];
                           j++;
                        }
                        k++; }

         /* Copy remaining elements of left array if any */
                       while (i <= leftEnd)
                             {
                                temp[k] = array[i];
                                i++;
                                k++;
                             }

        /* Copy remaining elements of right if any */


        for (i = leftStart; i <= rightEnd; i++)      // copy back
                arr[i] = temp[i];
}
```

mergeSort()

- We use a wrapper method to create the temporary array, and to make the initial call to a separate recursive method:

```
static void mergeSort(int[] arr) {
        int[] temp = new int[arr.length];
        myMergeSort(arr, tmp, 0, arr.length - 1);
}

static void myMergeSort(int[] arr, int[] temp, int start, int end) {
        if ( ? ) // base case
                return;
        int middle = (start + end)/2; // The splitting step

        ?

}
```

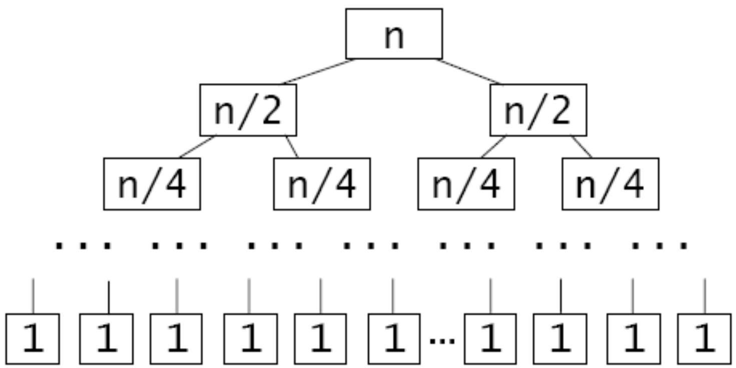
mergeSort()

```
static void mergeSort(int[] arr) {
        int[] temp = new int[arr.length];
        myMergeSort(arr, tmp, 0, arr.length - 1);
}

static void myMergeSort(int[] arr, int[] temp, int start, int end) {
        if (start >= end ) // base case
                return;
        int middle = (start + end)/2; // The splitting step

                        // Sort first and second halves
                          myMergeSort (arr, temp, start, middle);
                          myMergeSort (arr , temp, middle+1, end);

                          // Merge the sorted halves
                          merge(arr, temp, start, middle, middle+1, end);
```

Running Time Analysis

- Merging two halves of an array of size n requires 2n moves.
- Merge-sort repeatedly divides the array in half, so we have the following call tree:

recursive call tree for mSort() # moves



2n

2*2*(n/2) = 2n

4*2*(n/4) = 2n

...

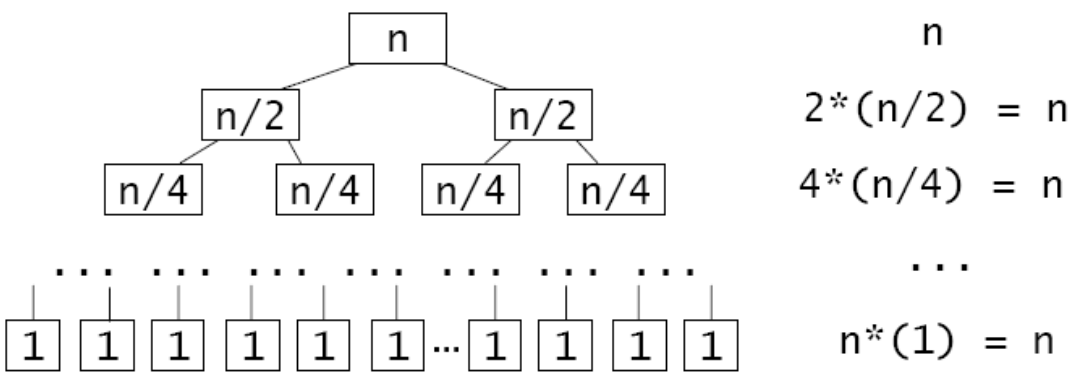
- At all but the last level of the call tree, there are 2n moves
  - How many levels are there?
- $M(n) = 2n\log_2 n$ (worst-case or best-case)
- $C(n) \leq n\log_2 n$ (between 0.5n*log(n) and n*log(n))
- O(n log n) overall

Compared to Quick-sort

- Partitioning an array requires n comparisons, because each element is compared with the pivot.

- ***best case:*** partitioning always divides the array in half

recursive call tree for qSort() # comparisons



  - at each level of the call tree, we perform n comparisons
  - There are $\log_2 n$ levels in the tree. So $C(n) = n\log_2 n$
  - $M(n) \leq 1.5\ n\log_2 n$ (at most n/2 swaps at each level)

Quick-Sort or Merge-Sort?

- Quick-sort used often
  - Low extra space
  - Good performance average
- For Quick-Sort
  - worst-case does not appear often, average-case is closer to best-case (n*log(n) comparisons and 1.5n*log(n) moves)
  - It is important to choose good pivots, to have n*log(n) running time
- For merge-sort
  - Average-case is close to worst-case
  - Between 0.5n*log(n) and n*log(n) comparisons and 2n*log(n) moves

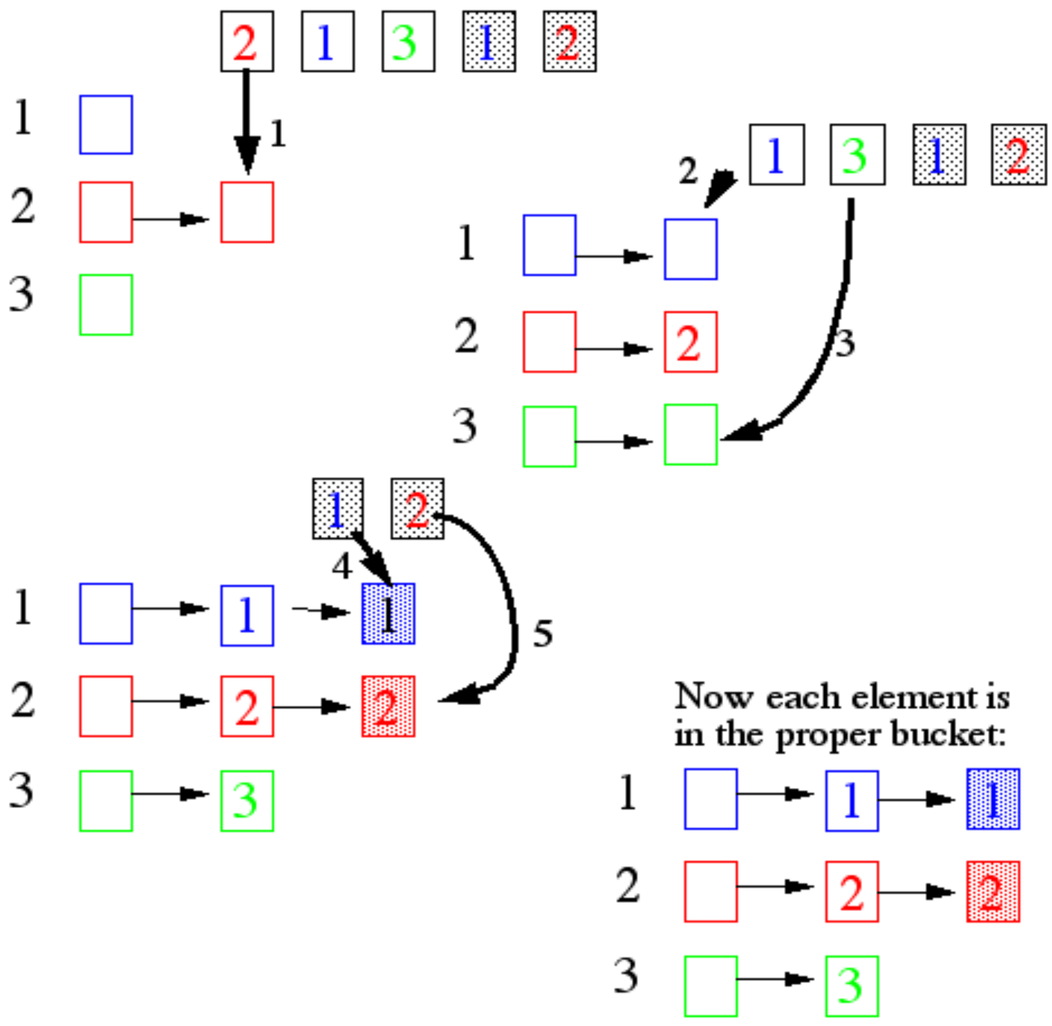
Comparisons

Algorithm	Best-case	Worst-case	Average-case	Extra space
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell-sort	$O(n*logn)$	$O(n^{1.5})$	$O(n^{1.5})$, but can be $O(n^{1.25})$	$O(1)$
Quick-sort	$O(n*logn)$	$O(n^2)$	$O(n*logn)$	$O(logn)$ (ave case) $O(n)$ (worst case)
Merge-sort	$O(n*logn)$	$O(n*logn)$	$O(n*logn)$	$O(n)$
Heap-sort	$O(n*logn)$	$O(n*logn)$	$O(n*logn)$	$O(1)$

Bucket Sort

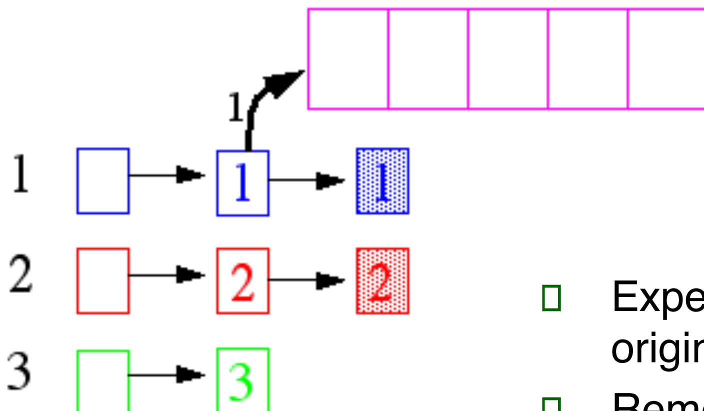
- Bucket sort
  - Assumption: integer keys in the range [0, M)
  - Basic idea:
    1. Create *M* linked lists (*buckets*), one for each possible key value
    2. Add each input element to appropriate bucket
    3. Concatenate the buckets
- Remember hash tables also uses buckets?

Bucket Sort Example

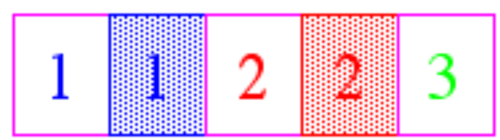


Bucket Sort Example

- Pull the elements from the buckets into the array



- Expected total time is O(M+N), with N = size of original array, M = number of buckets
- Remember hash tables also uses buckets?
  - Bucket sort preserves order (key values) in buckets
  - Hashing mixes up elements with diverse key values



Keys of Non-integer Types?

- What if keys are not integers?
  - Assumption: input is *N* floating numbers (scaled) in [0, 1]
  - Basic idea:
    - Create *M* linked lists (*buckets*) to divide interval [0,1] into subintervals of size 1/*M*
    - Add each input element to appropriate bucket and sort the bucket with insertion sort
  - Choose M=O(N)
  - Uniform input distribution → expected bucket size is O(1)
    - Therefore the expected total time is O(N): O(N) to put elements into the buckets and O(N) to move them back into the array

- With uniform key distribution, Bucket Sort has O(N) running time
- But sensitive to the key distribution in the range (what if it's not uniform?)
- Pays with space for time
- Pays with worst-case time for average-case time