Merge-Sort

EECS 233

Previous Lecture: Quick-Sort

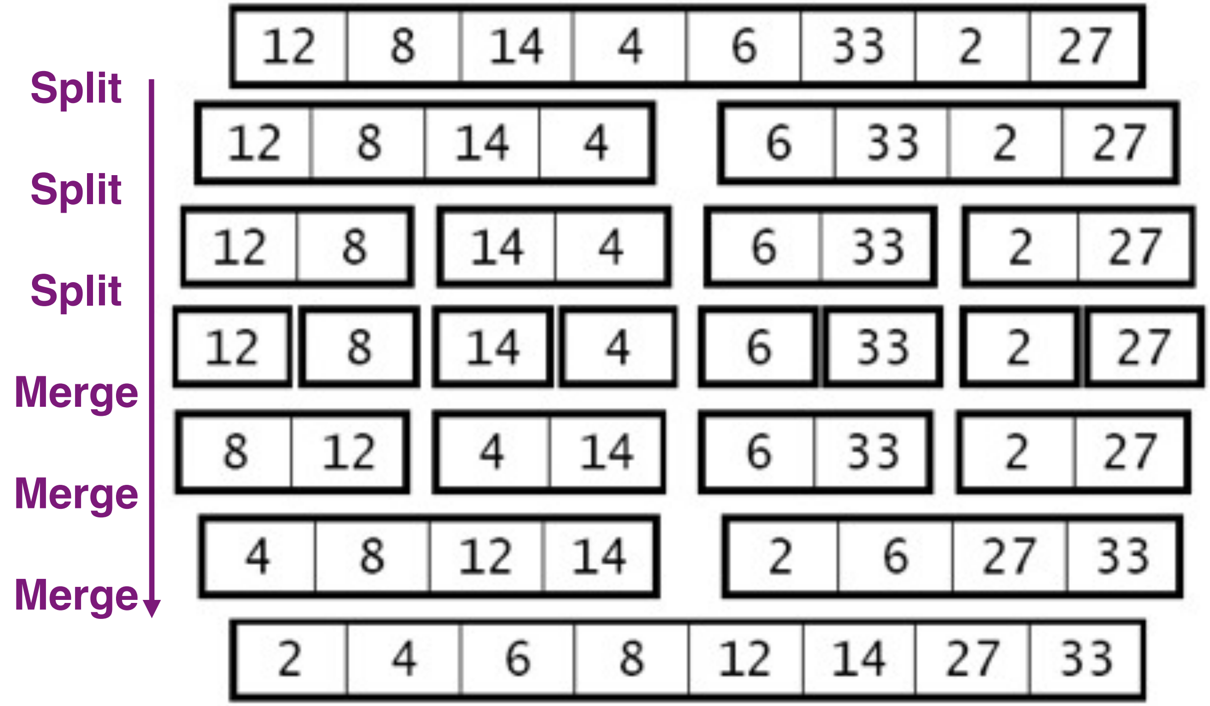
- Quick-Sort: a recursive, divide-and-conquer algorithm:
  - *divide:* partition the array into two subarrays so that :
    - ✓ *each element in the left array <= each element in the right array*
  - *conquer:* apply quick-sort recursively to the subarrays, stopping when a subarray has a single element
  - *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 $O(n \log n)$ and worst-case $O(n^2)$

Merge-Sort

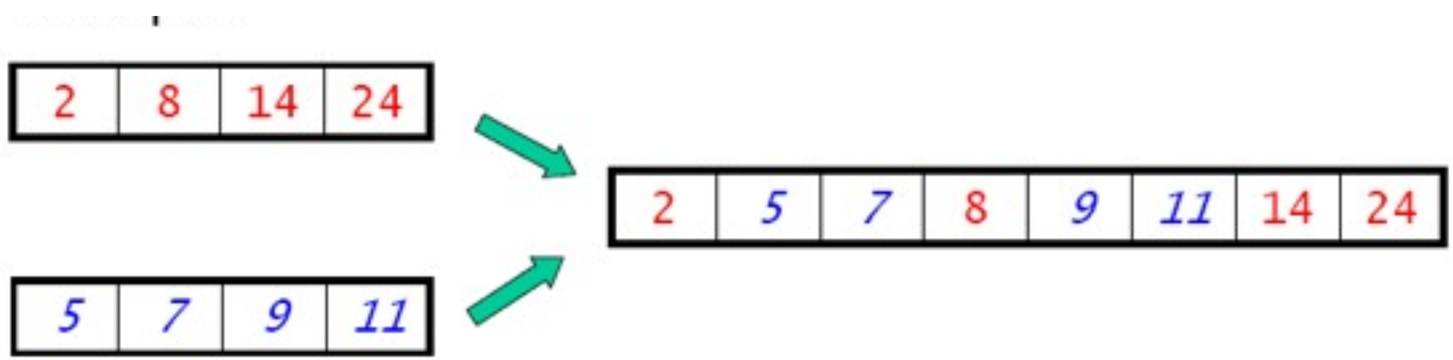
- Like quick-sort, merge-sort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply merge-sort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays



Merge-Sort

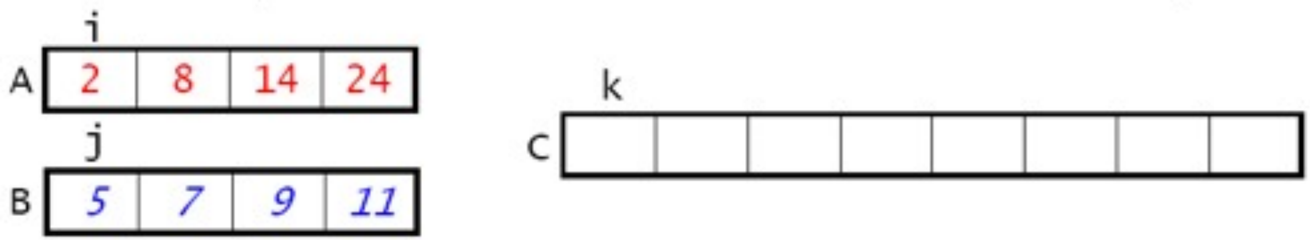
- All of the sorting algorithms we've seen thus far have sorted the array in place. They used only a small amount of additional memory, i.e., O(log n) additional space (for recursion)

- 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.

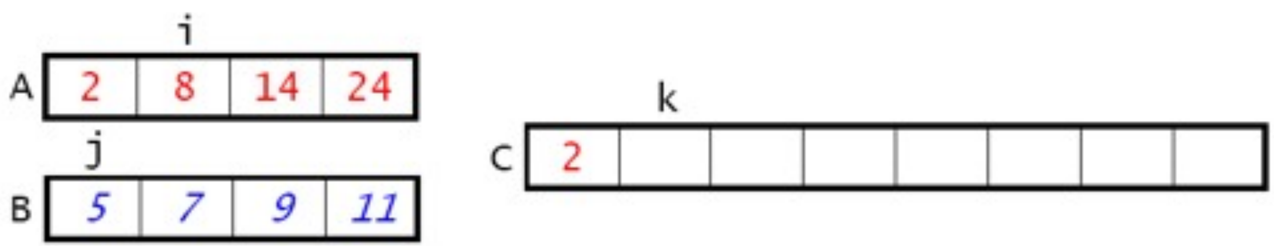


Merging Sorted Subarrays

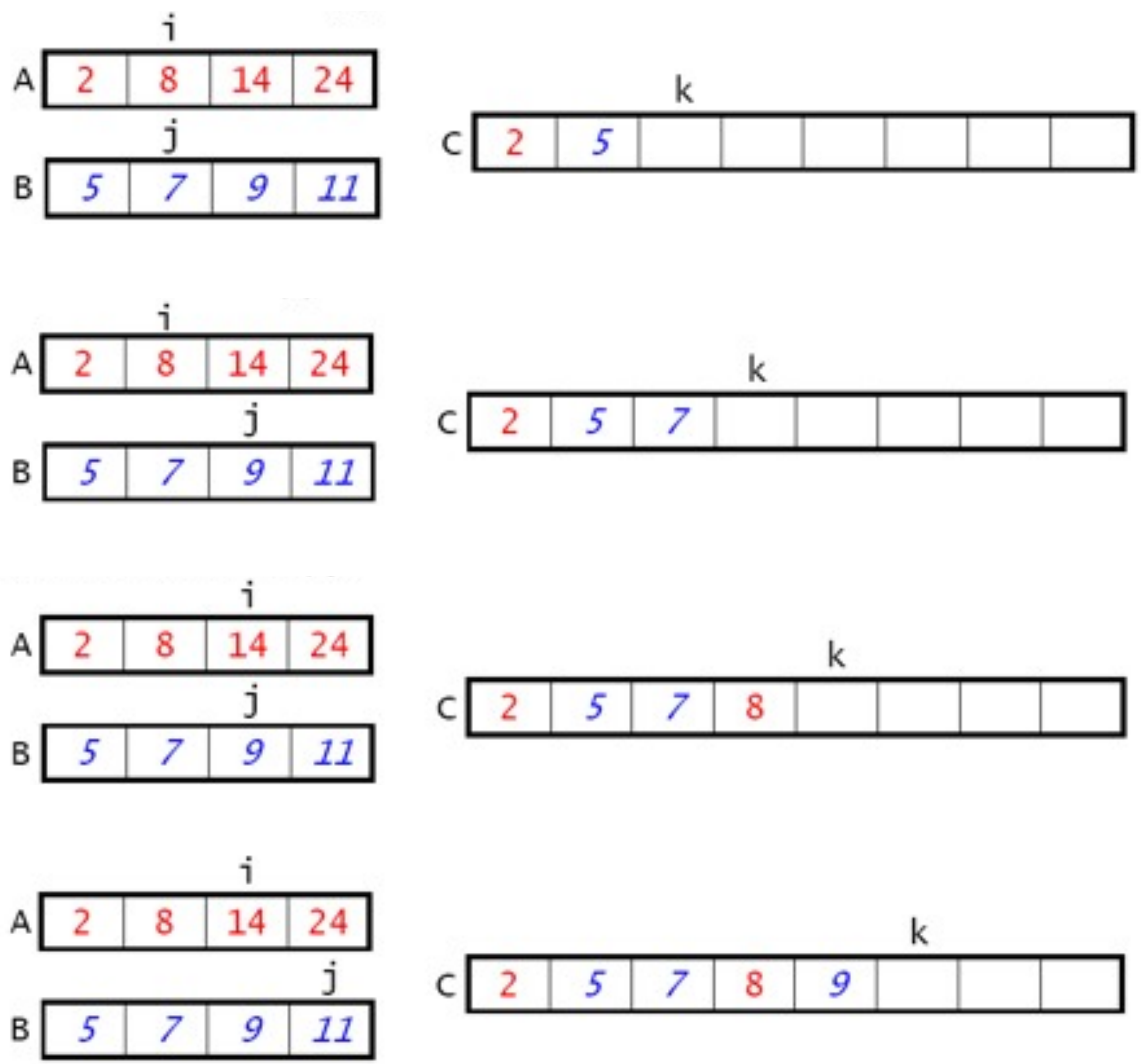
- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:



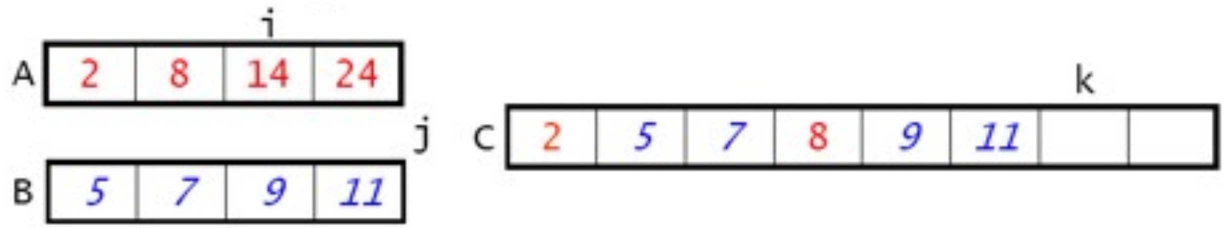
- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

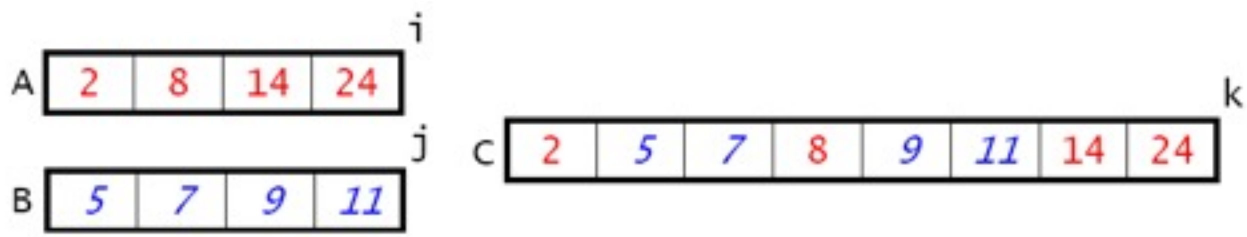


Merging Sorted Subarrays - Steps



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

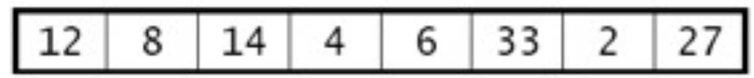
Recursive Procedure - Skeleton

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

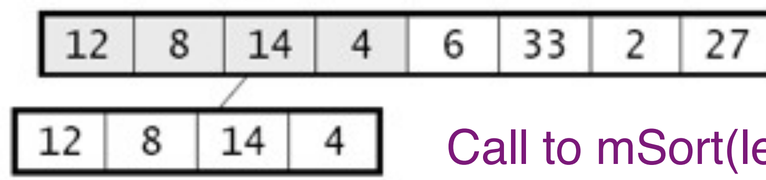
```
static void mergeSort(int[] arr, int length) {
        mSort(arr);
}

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

Recursive Calls - Steps

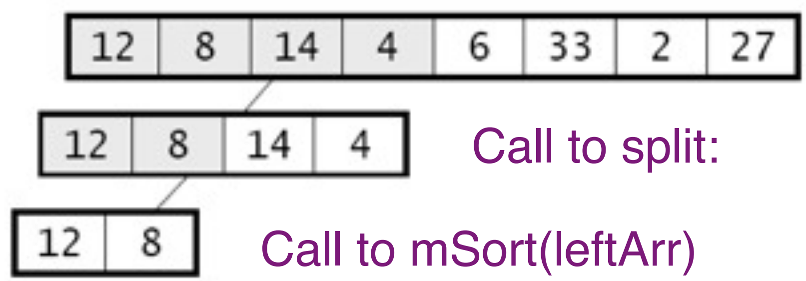
```
static void mergeSort(int[] arr, int length) {
        mSort(arr);
}

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



Call to split:



Call to mSort(leftArr)



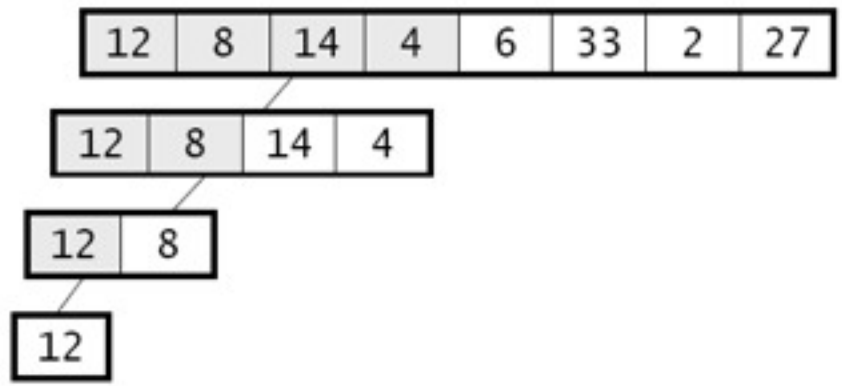
Call to split:

Call to mSort(leftArr)

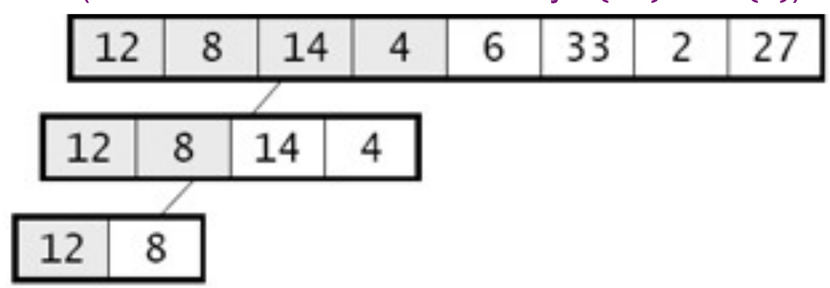
Recursive Calls - Steps

```
static void mergeSort(int[] arr, int length) {
        mSort(arr);
}

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

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



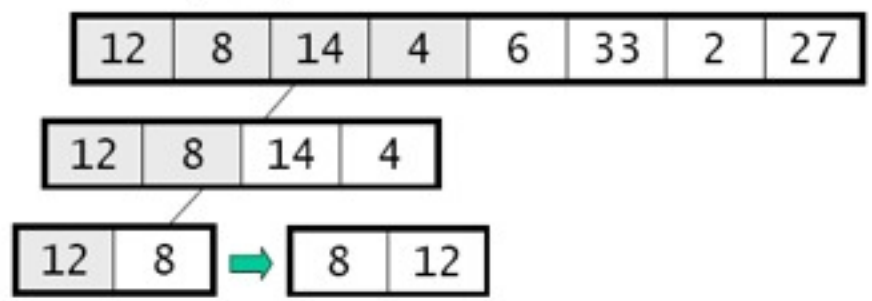
We are down to the base cases, so simply return (we have two sorted subarrays {12} and {8})



Recursive Calls - Steps

```
static void mergeSort(int[] arr, int length) {
        mSort(arr);
}

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

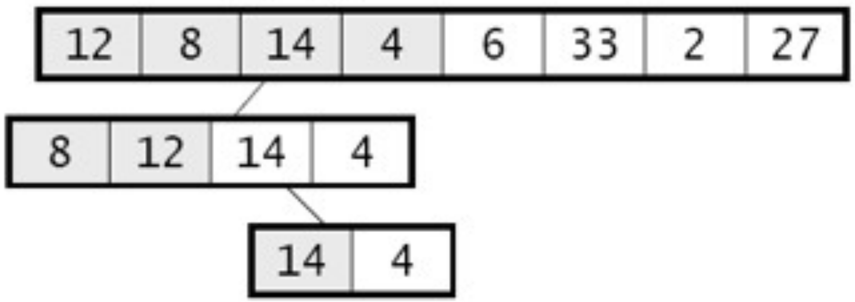
Call merge() to merge two subarrays into original array

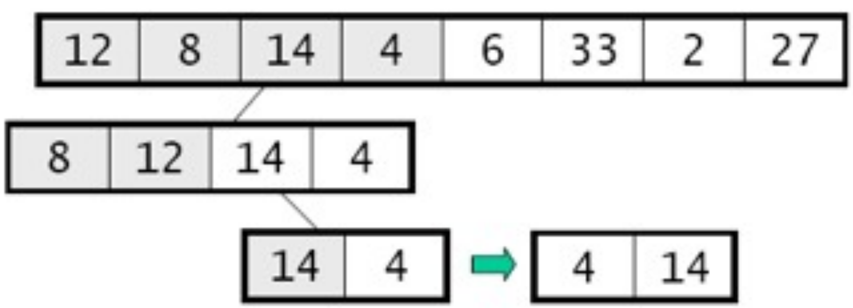


Return to the recursive call for the 4-element subarray, and start another recurise call for the right subarray {14, 4}.

Recursive Calls - Steps

Repeat the similar process for the right 2-element subarray

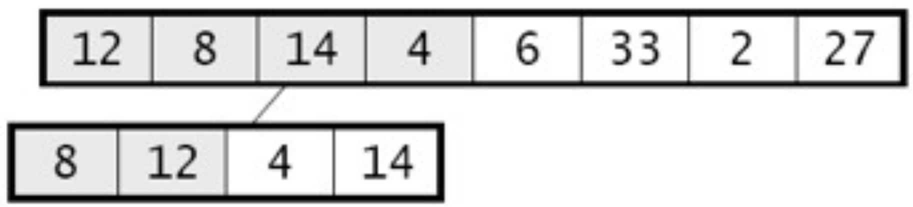




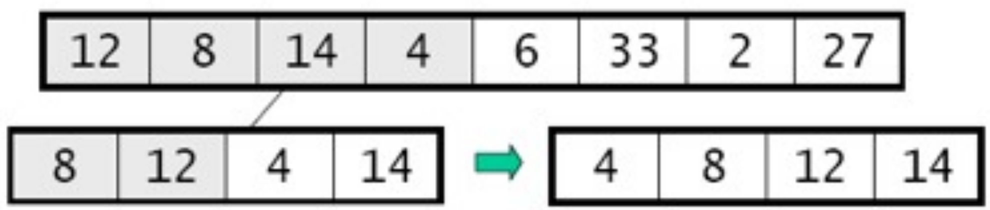

```
static void mergeSort(int[] arr, int length) {
        mSort(arr);
}

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

Recursive Calls - Steps

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

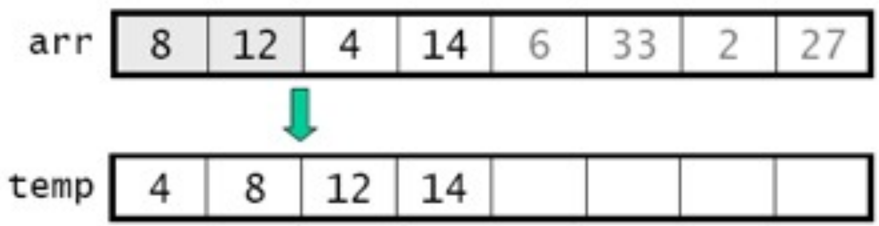


Call merge() to merge the 2-element subarrays, and copy the elements back

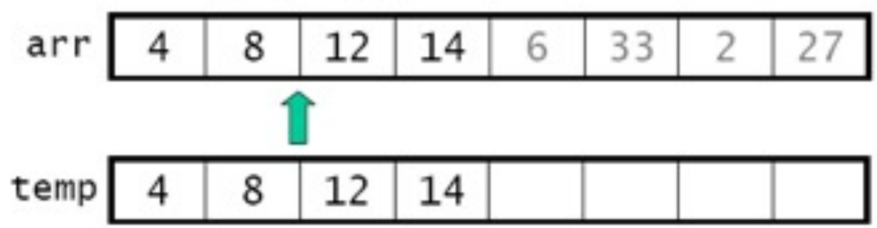


Implementation of Merge-Sort

- 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:

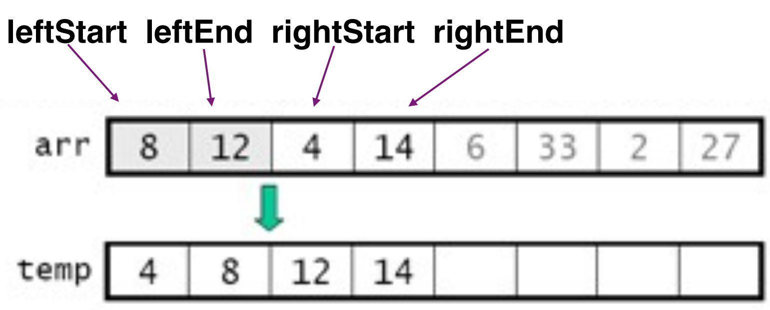


  - after each merge, copy the result back into the original array:



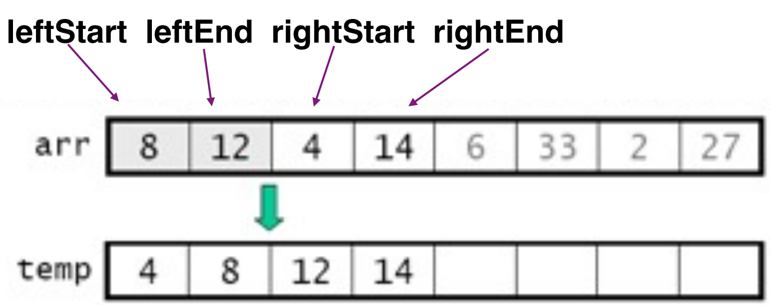
The Helper Method merge()

```
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 ];
}
```



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 ( i <= leftEnd && j <= rightEnd ) {
                if (arr[ i ] < arr[ j ])
                    temp[ k++ ] = arr[ i++ ];
                else
                    temp[ k++ ] = arr[ j++ ];
        }
        while ( i <= leftEnd)
            temp[ k++ ] = arr[ i++ ];

        while (j <= rightEnd)
            temp[ k++ ] = arr[ j++ ];

        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 length) {
        int[] temp = new int[length];
        mSort(arr, tmp, 0, length - 1);
}

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

        ?



}
```

mergeSort()

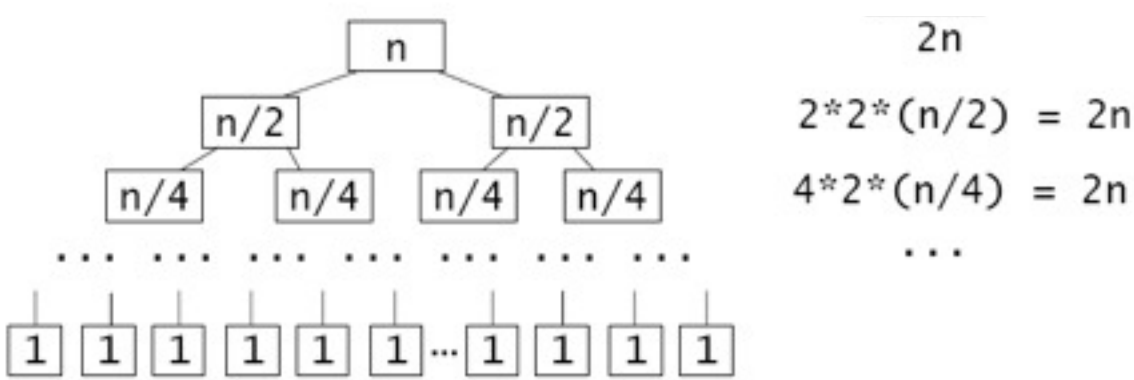
- We use a wrapper method to create the temporary array, and to make the initial call to a separate recursive method:

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

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

        mSort( arr, temp, start, middle );
        mSort( arr, temp, 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

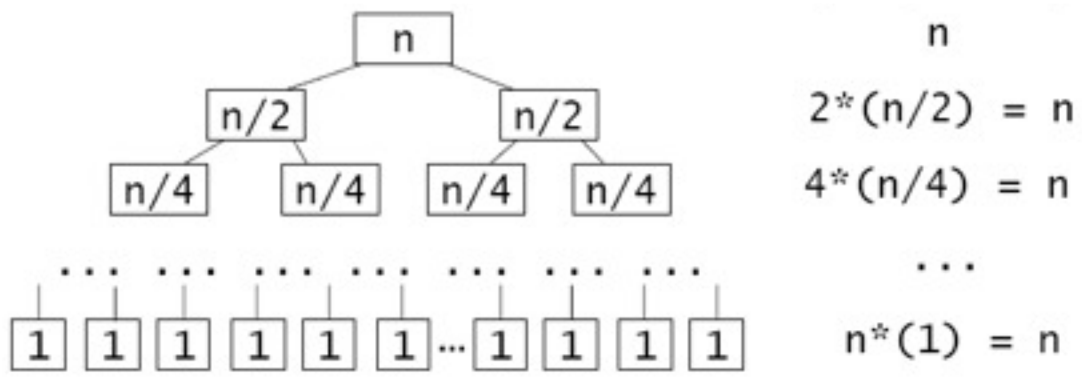


  - At all but the last level of the call tree, there are 2n moves
    - ✓ How many levels are there?
  - M(n) = ? C(n) = ?
  - Worst-case or best-case

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 $C(n) = n\log_2 n$
- $M(n) \sim 1.5\, n\log_2 n$

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
  - < n*log(n) comparisons and 2n*log(n) moves

Exercises Merge-sort and Quick-sort

Follow lecture, work through exercises, count C(n) and M(n):

12 15 24 16 11 3 5 21 8 14

Question of the week:

Can we avoid copying merged arrays back from temp to the original array