Lecture 7: Chapter 2 part 2

Divide-and-Conquer

CS3310

Binary Search

- Binary search can be implemented as a recursive divide-and-conquer algorithm.

Problem: Is x in the *sorted* array S?

If x equals the middle item, return true. Otherwise:

1. *Divide* the array into two subarrays about half as large. If x is smaller than the middle item, choose the left subarray. If x is larger than the middle item, choose the right subarray
2. *Conquer* the subarray by determining whether x is in that subarray. We do this recursively.
3. *Obtain* the solution to the top-level call from the solution to the recursive call.

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 25 27 30 35 40 45 47

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 **25** 27 30 35 40 45 47

The middle item is 25, which is not what x equals.

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 25 **27 30 35 40 45 47**

- We then divide the array into 2 subarrays, one to the left and one to the right of the middle item we checked.
- Since $18 < 25$, we *conquer* the left subarray by passing it to a recursive call to `BinarySearch`.

Binary Search Top-Level Description

If $x = 18$, and we have the following array, we first see if the middle element equals x

10 12 13 14 18 20 25 27 30 35 40 45 47

- ➢ We conquer the left subarray by determining if 18 is in it.
- ➢ Since 18 **is** in this left subarray, we return true to the top-level call to `BinarySearch`, which then returns true.

Note: Although `BinarySearch` would take another recursive step in this problem (18 isn't the middle item of the left subarray), we are currently describing what happens at the top level (i.e. recursively call `BinarySearch` on a subarray and wait for a response).

Binary Search In-Depth Description

1. Check the middle value:

10 12 13 14 18 20 **25** 27 30 35 40 45 47

2. 18 != 25. Divide into 2 subarrays:

10 12 13 14 18 20 25 **27 30 35 40 45 47**

3. 18 < 25, perform BinarySearch on the *left* subarray. Check the middle value:

10 12 **13** 14 18 20 25 27 30 35 40 45 47

4. 18 != 13. Divide into 2 subarrays:

10 12 13 **14 18 20** 25 27 30 35 40 45 47

5. 18 > 13, perform BinarySearch on the *right* subarray. Check the middle value:

10 12 13 14 **18** 20 25 27 30 35 40 45 47

6. 18 = 18, value found! Return true.

Binary Search Pseudocode

Problem: Determine whether *x* is in the sorted array *S* of size *n*

Parameters: Positive integer *n*, sorted array of keys *S* indexed from 1 to *n*, a key *x*

Outputs: The location of x in S, or 0 if x is not in S.

```
index BinarySearch(index low, index high)     // low = 1 and high = n at top level
     if (low > high)
          return 0;                           // key not found in array
     else
          index middle = ⌊(low + high) / 2⌋
          if (x == S[middle])
               return middle
          else if (x < S[middle])
               return BinarySearch(low, middle - 1)
          return BinarySearch(middle + 1, high)
```

Binary Search Analysis

```
index middle = ⌊(low + high) / 2⌋
if (x == S[middle])
     return middle
else if (x < S[middle])
     return BinarySearch(low, middle - 1)
return BinarySearch(middle + 1, high)
```

What is the basic operation?

Binary Search Analysis

```
index middle = ⌊(low + high) / 2⌋
if (x == S[middle])
     return middle
else if (x < S[middle])
     return BinarySearch(low, middle - 1)
return BinarySearch(middle + 1, high)
```

What is the basic operation? Although we have `x == S[middle]` and `x < S[middle]`, we can count this as <u>one</u> operation.

➢ Remember, we always assume comparisons are implemented as efficiently as possible. Here, we assume that in assembly an if/else-if requires a single comparison.

Binary Search

- The recursive version of `BinarySearch` employs **tail-recursion**.
    - No operations are performed after the recursive call.
- For this reason, developing an iterative version of `BinarySearch` is simple.
- We discussed a recursive version of this algorithm since it clearly illustrates the divide-and-conquer process of dividing an instance into smaller instances.
- However, in many languages (such as C++) it is beneficial to use an iterative approach over a recursive one.
    - Why?

Binary Search

- The recursive version of `BinarySearch` employs **tail-recursion**.
  - No operations are performed after the recursive call.
- For this reason, developing an iterative version of `BinarySearch` is simple.
- We discussed a recursive version of this algorithm since it clearly illustrates the divide-and-conquer process of dividing an instance into smaller instances.
- However, in many languages (such as C++) it is beneficial to use an iterative approach over a recursive one.
  - Why?
  - An activation record is pushed to the call stack for each recursive call.
    - Removing the need to add these to the call stack increases both efficiency and memory usage.

In-Class Exercise

1. The comparison in the `else` statement is the basic operation. Analyze `BinarySearch` by finding its recurrence relation and determining its best-case and worst-case order.

```
index BinarySearch (index low, index high)          // low = 1 and high = n at top level
    if (low > high)
        return 0;                                    // key not found in array
    else
        index middle = ⌊(low + high) / 2⌋
        if (x == S[middle])
            return middle
        else if (x < S[middle])
            return BinarySearch(low, middle - 1)
        return BinarySearch(middle + 1, high)
```

2. Design an iterative version of the `BinarySearch` algorithm.

Merge Sort

Problem: Given a list S of n numbers, sort them in nondecreasing order

1. **Divide**: Split the list into two sublists of similar size.
2. **Conquer**: Recursively sort each sublist.
3. **Obtain**: Merge the two sorted lists into a complete sorted list.

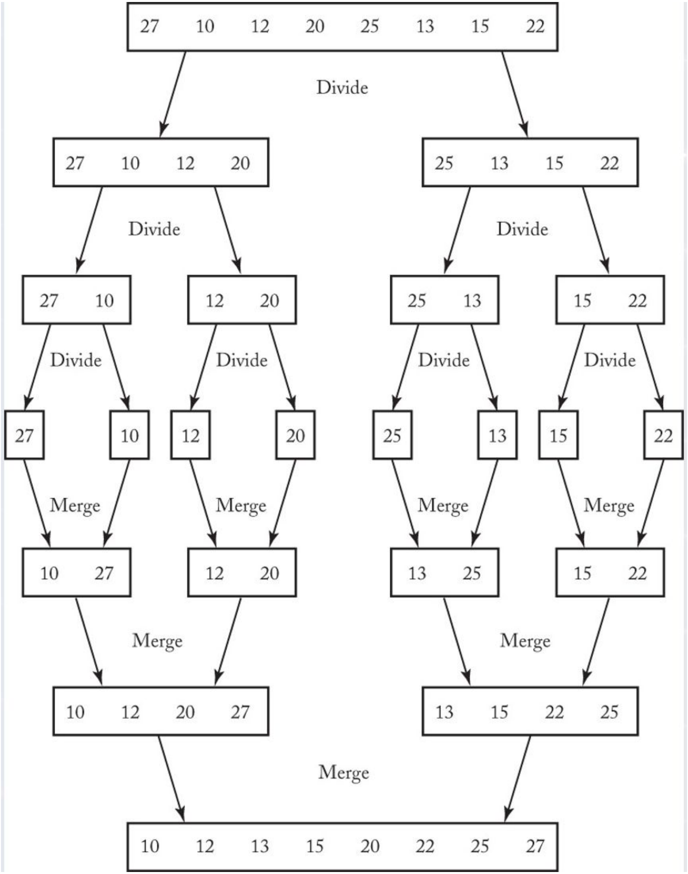
High-Level Example:

S = [4, 5, 1, 7, 8, 10, 2, 5]

1. **Divide**: [4, 5, 1, 7] and [8, 10, 2, 5]
2. **Conquer**: [1, 4, 5, 7] and [2, 5, 8, 10]
3. **Obtain**: Merge the sublists by placing each item in a new list in the correct order: [1, 2, 4, 5, 7, 8, 10]

Merge Sort Complete

- This image shows the steps taken by a human when performing a `MergeSort`
- Each sublist is split and `MergeSort` is recursively called until we arrive at lists of one item.
- A list of one item is already sorted, so this is the base case.
- When we have two items to merge, such as 10 and 27, we simply put the smaller value first and the bigger value second.



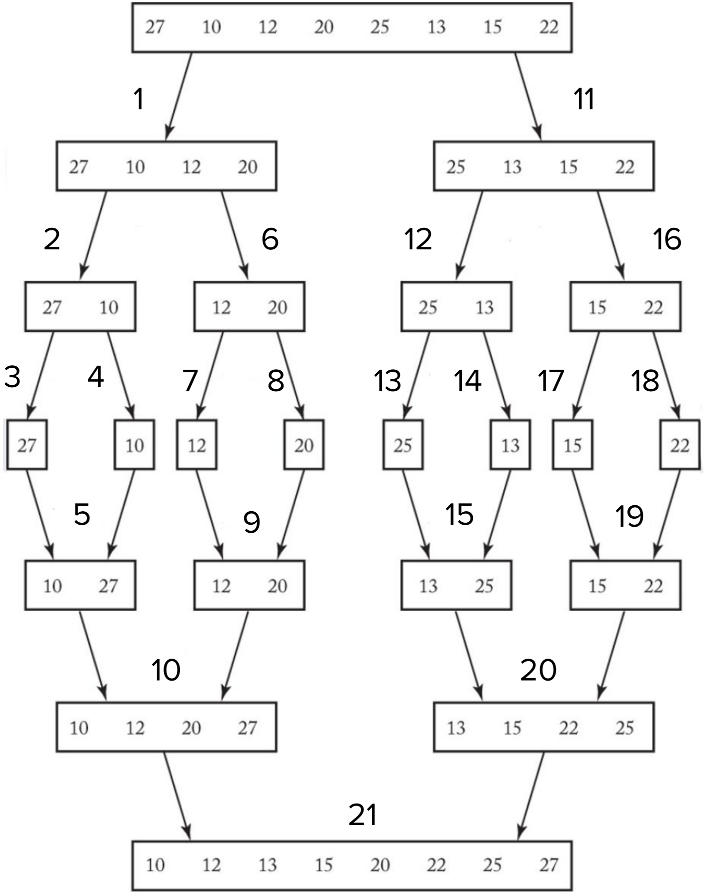
Merge Sort Pseudocode

```
void mergeSort(int n, keytype S[])
     if (n > 1)
          const int uLen = ⌊n/2⌋, vLen = n - uLen;
          keytype U[1..uLen], V[1..vLen];
          copy S[1] through S[uLen] to U[1] through U[uLen]
          copy S[uLen+1] through S[n] to V[1] through V[vLen]
          mergeSort(uLen, U)
          mergeSort(vLen, V)
          merge(uLen, vLen, U, V, S)
```

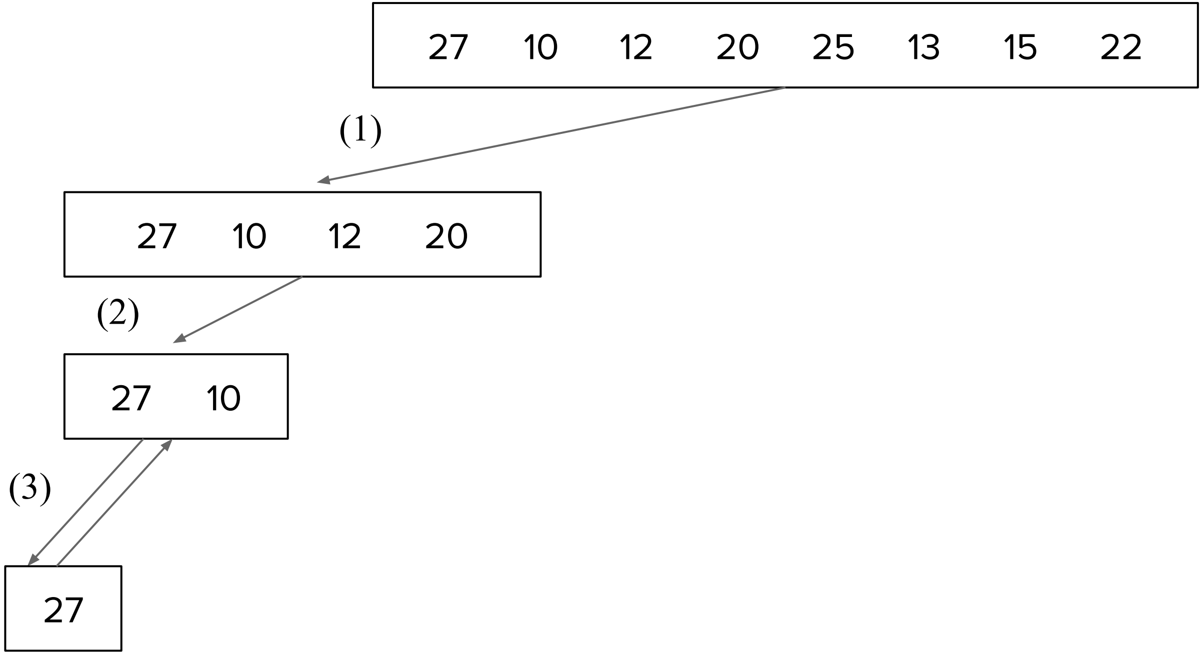
- In each call to `MergeSort`, we create two new arrays *U* and *V*.
- The left half of *S* is placed in *U* and the right half is placed in *V*.
- `MergeSort` is then called on *U* and *V*.
- Once both recursive calls return, *U* and V have been sorted. The final call to `merge` merges the items in *U* and *V* into *S*.

Merge Sort Complete

- This image shows the ordering of steps taken by a computer when performing a `MergeSort`
- As with `MinMax`, we recurse to the left until we reach a base case. (i.e. step 3)
- We then pop up a level and recurse to the right. If we reach a base case, we pop up a level again (i.e. step 4)
- Once both recursive calls return, the results are merged together (i.e. step 5)

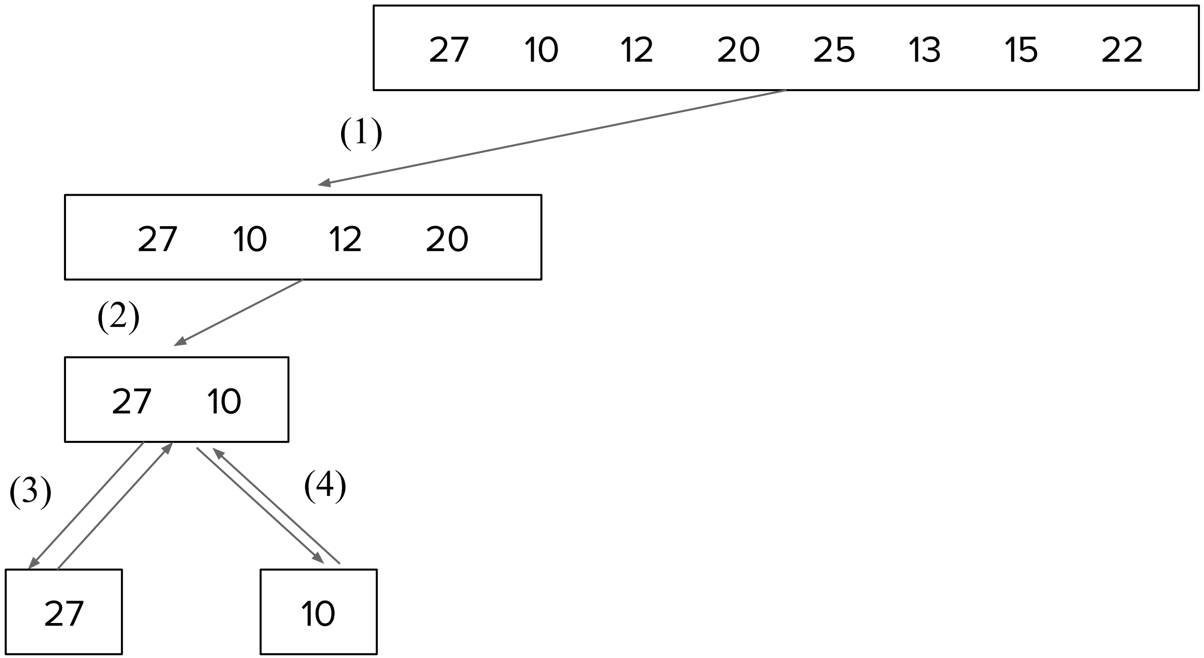


Merge Sort Complete



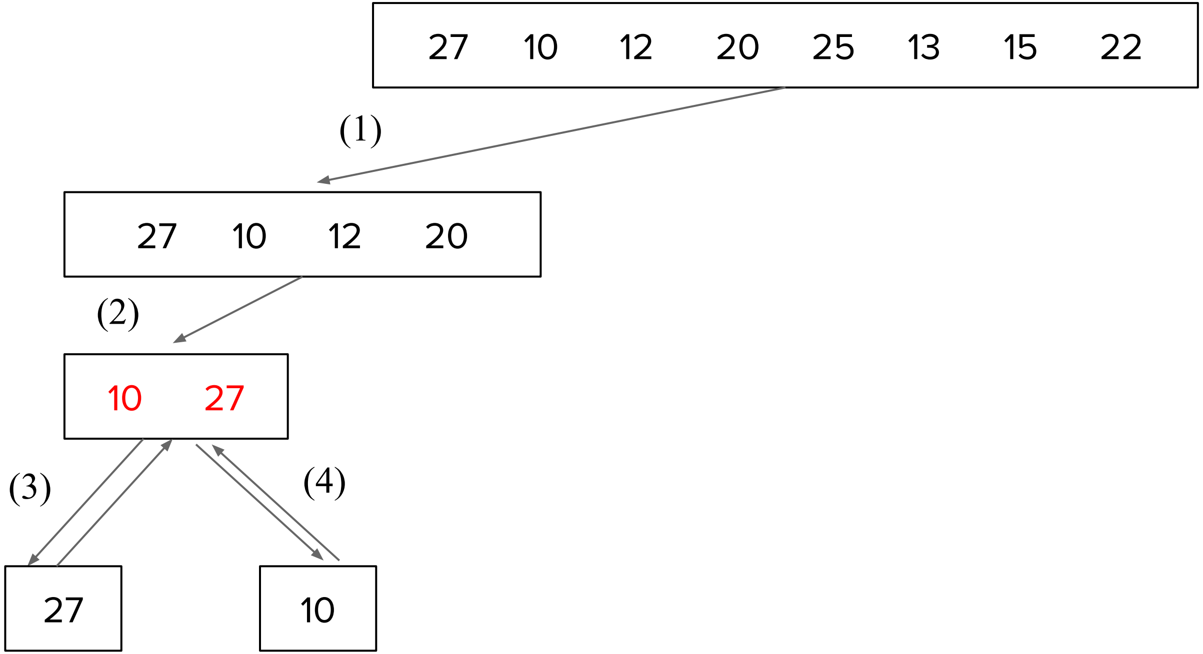
- We recurse three times to the left and reach a base case.
- We do nothing at a base case except pop back to the previous level of recursion.

Merge Sort Complete



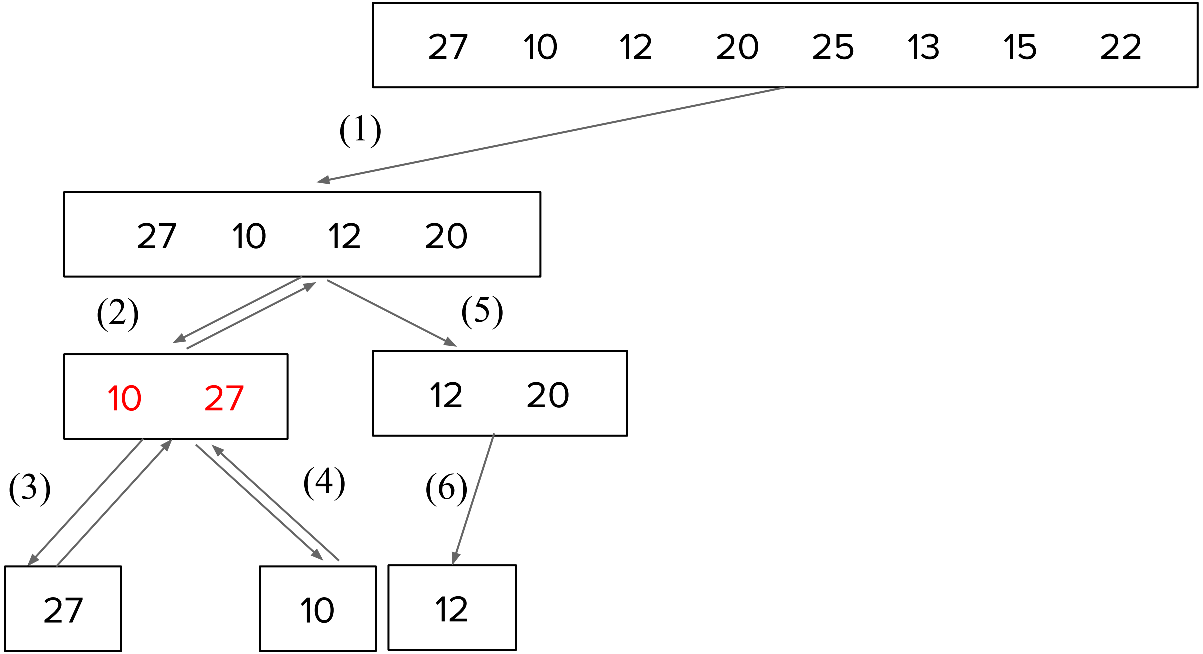
- When we return to the second level of recursion, we recurse to the right.
- We reach another base case and immediately pop back to the second level.

Merge Sort Complete



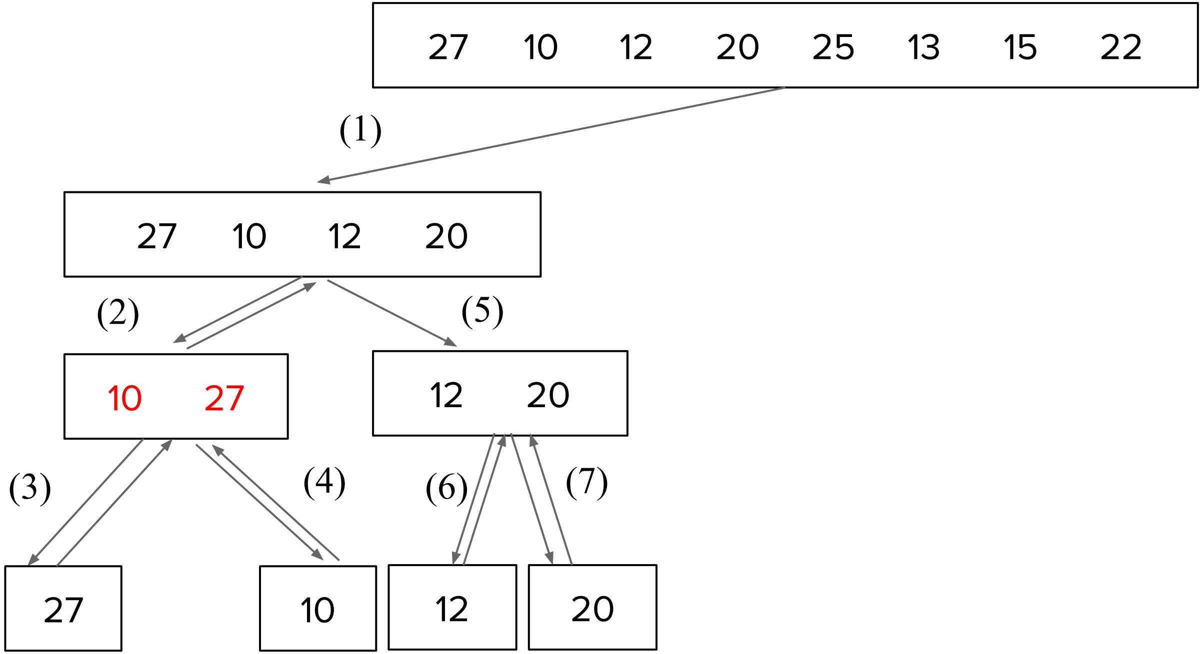
- The second level of recursion has completed both of its recursive calls.
- `Merge` is called on U and V and the results are placed in the second level's S array.

Merge Sort Complete



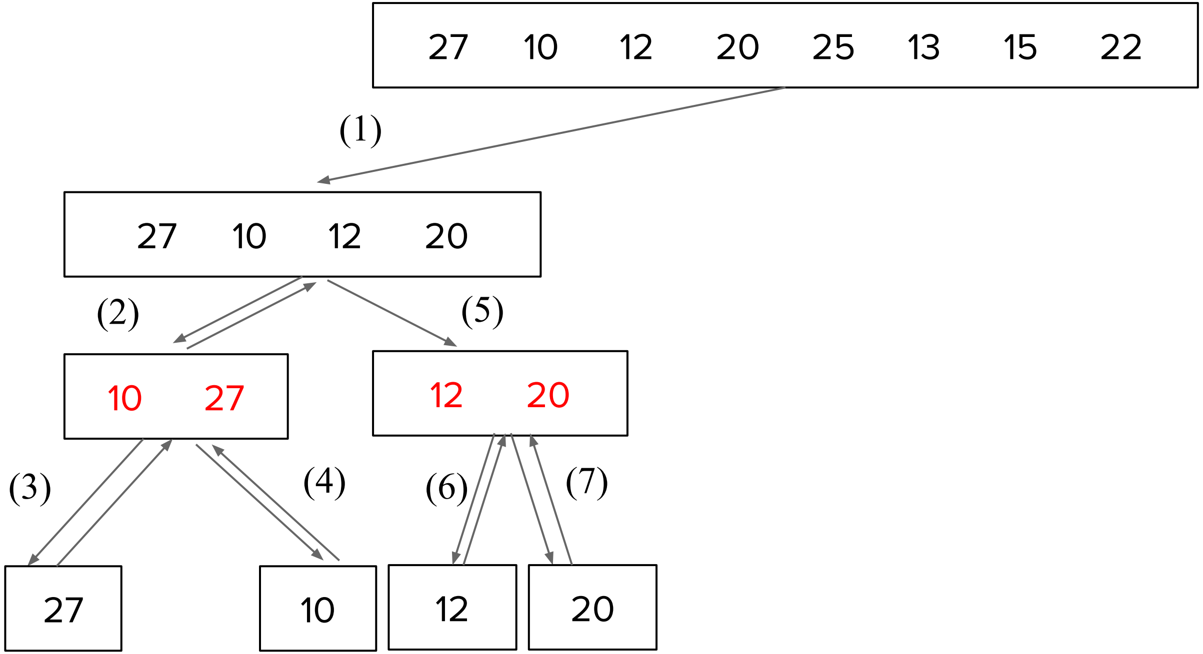
- We pop back to the first level of recursion. This level's first recursive call is finished so we recurse right (5). We are not at a base case so we recurse left (6).

Merge Sort Complete



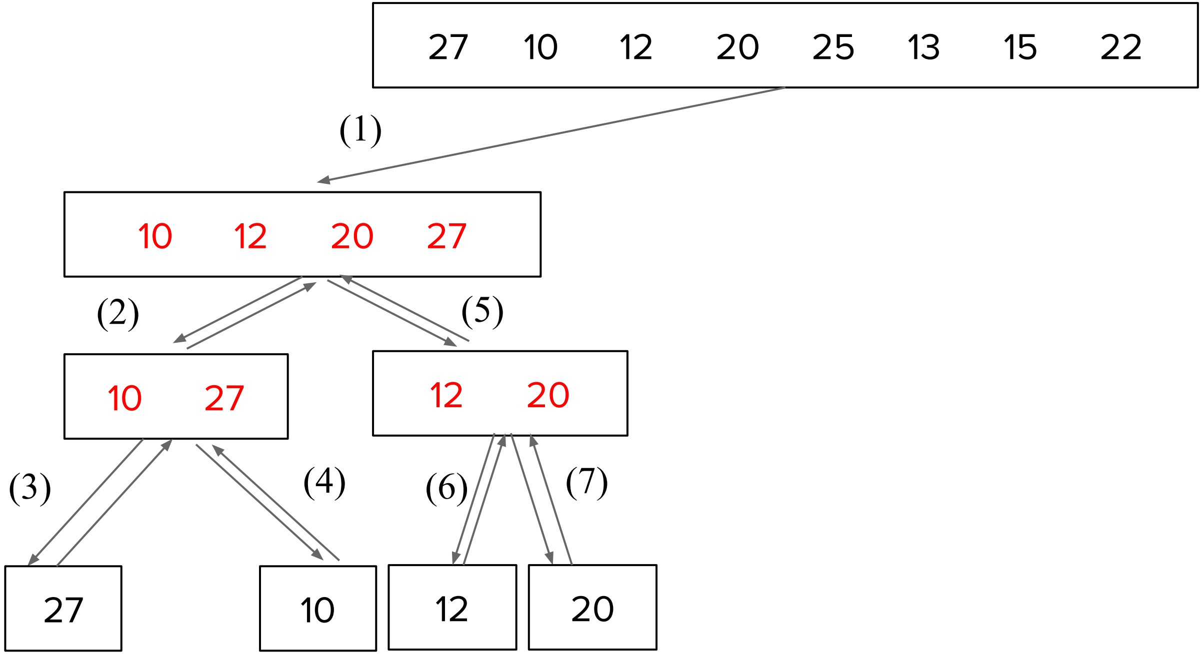
- We pop back to the second level of recursion and recurse right.
- We immediately reach another base case and pop back up to the second level.

Merge Sort Complete



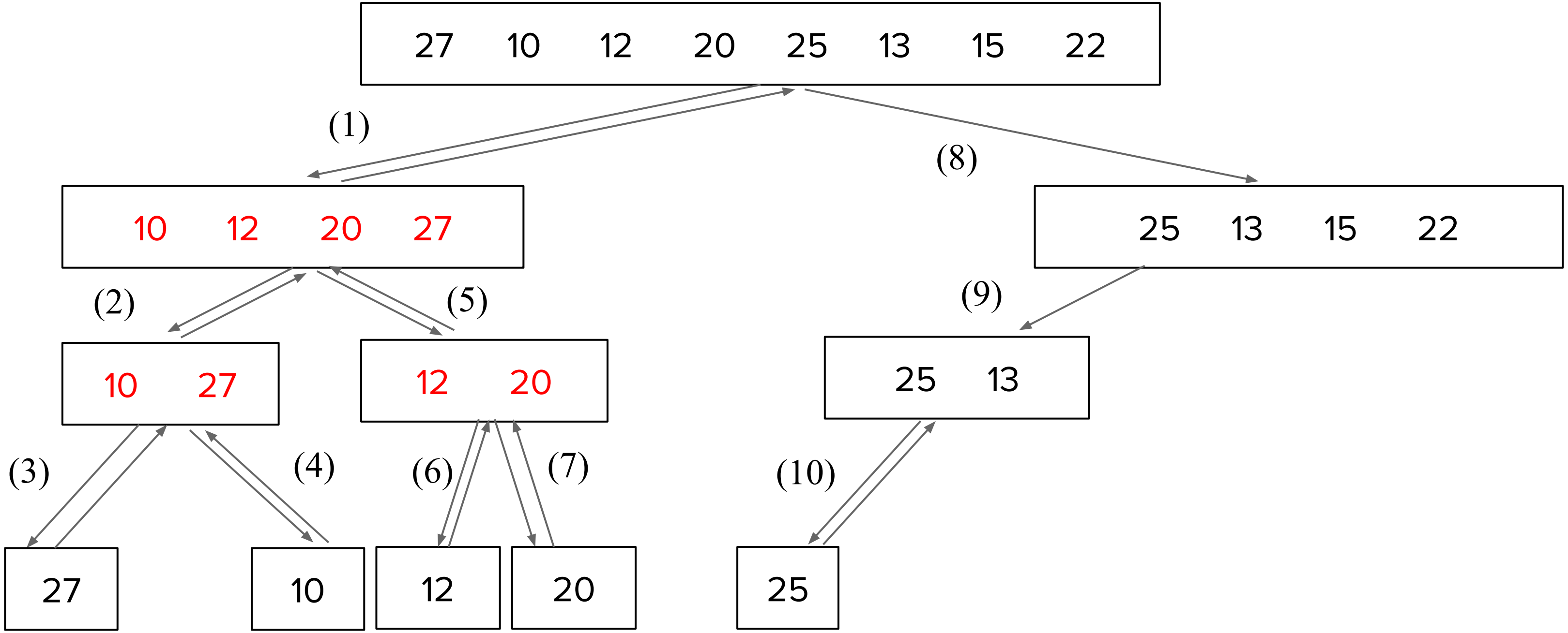
- The second level of recursion has completed both of its recursive calls.
- Merge is called on U and V and the results are placed in the second level's S array.

Merge Sort Complete



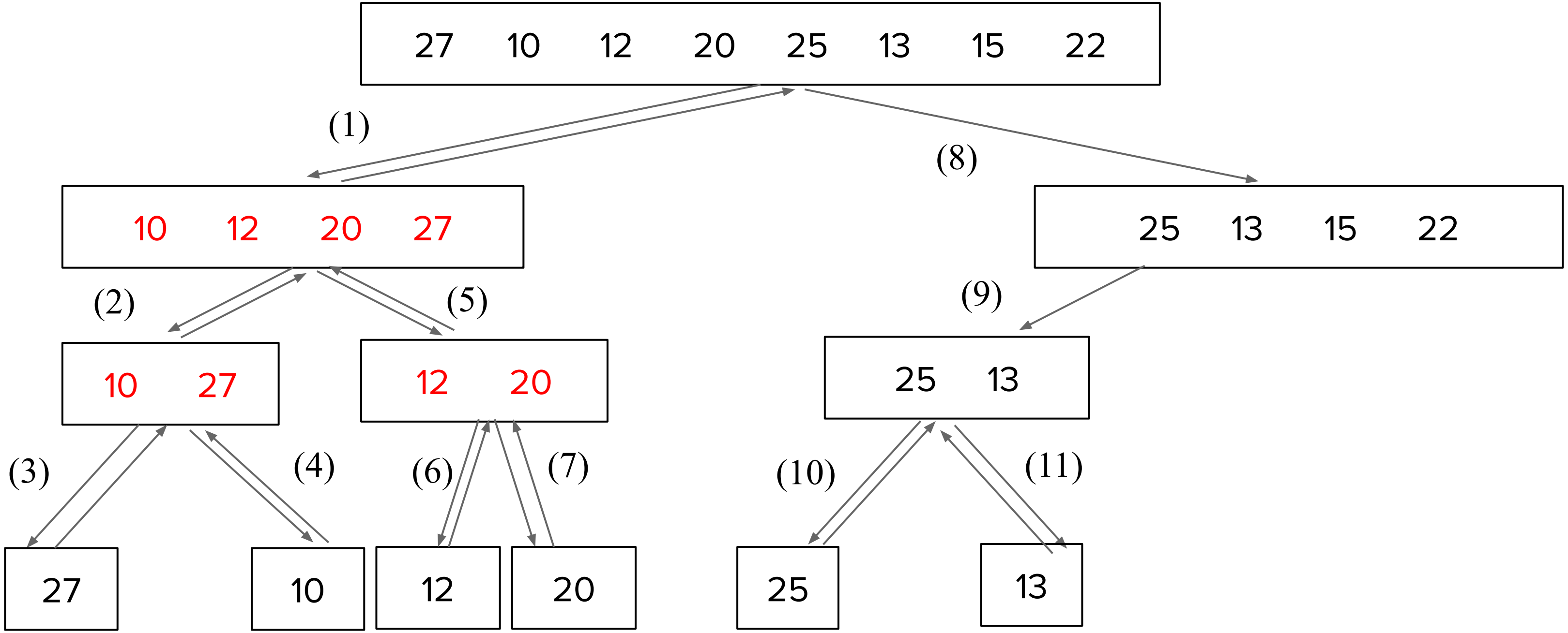
- The first level of recursion has completed both of its recursive calls.
- Merge is called on this level's U and V and the results are placed in the first level's S array.

Merge Sort Complete



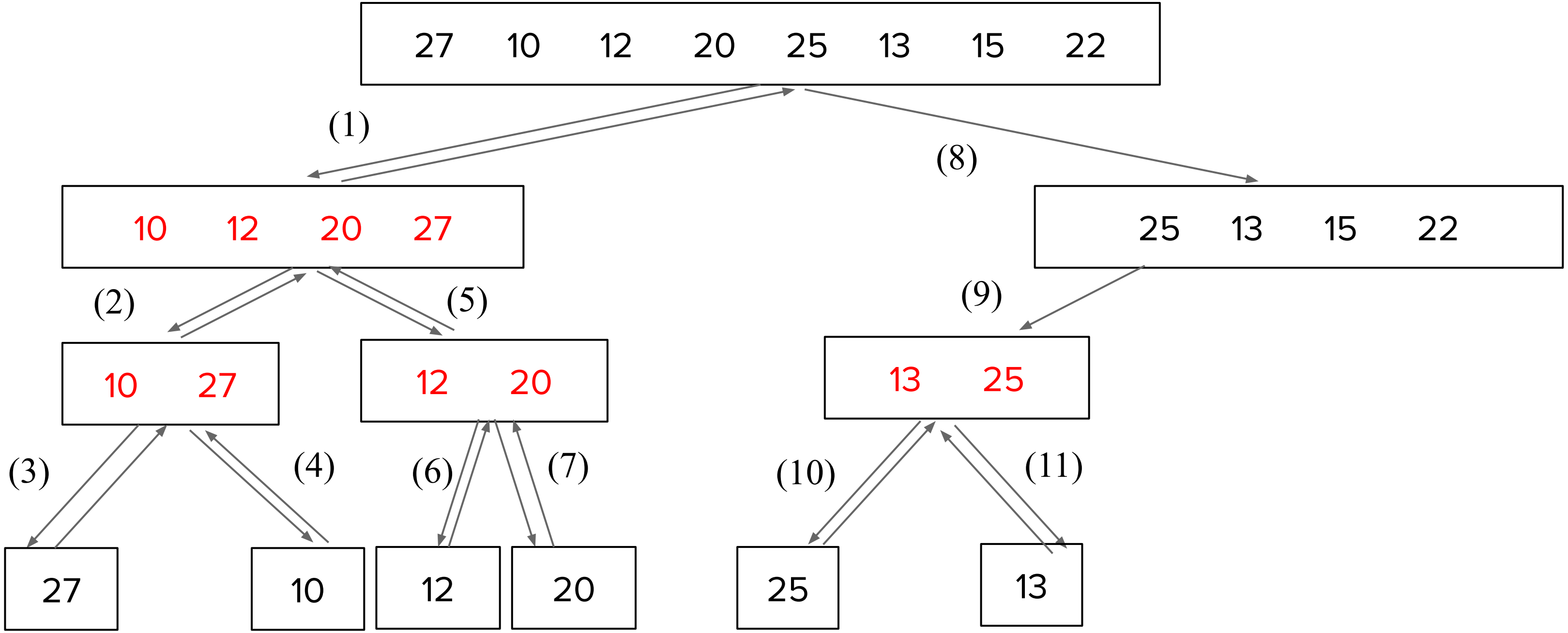
- We pop back to the top level and recurse right.
- From this first level of recursion, we recurse to the left until we reach a base case.

Merge Sort Complete



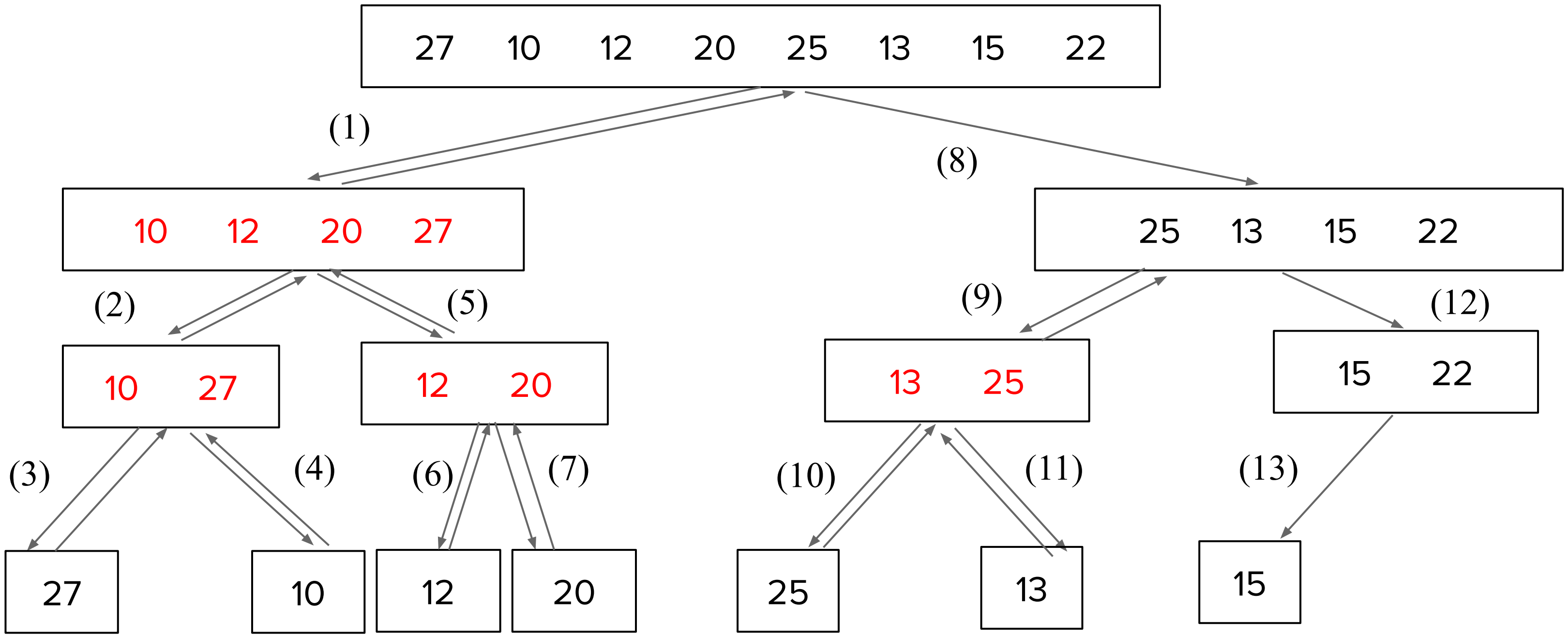
- When we return to the second level of recursion, we recurse to the right.
- We reach another base case and immediately pop back to the second level.

Merge Sort Complete



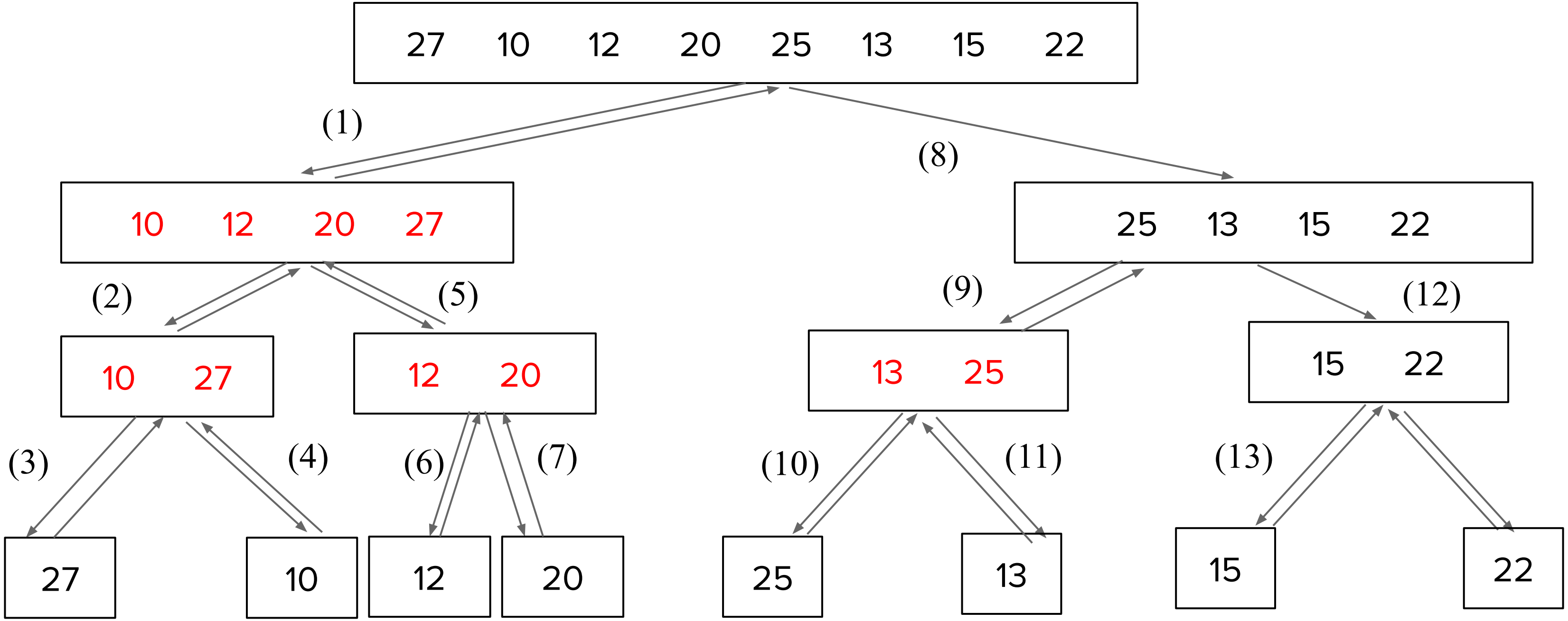
- The second level of recursion has completed both of its recursive calls.
- Merge is called on U and V and the results are placed in the second level's S array.

Merge Sort Complete



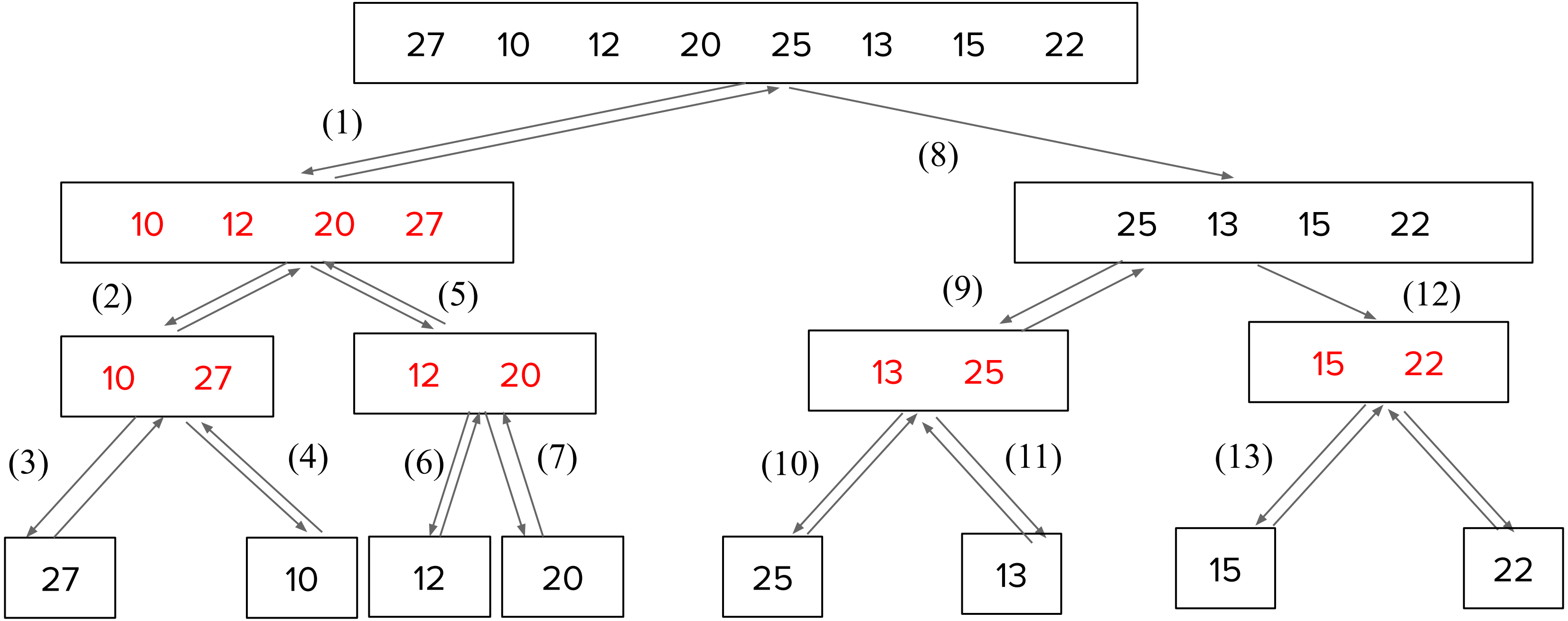
- We pop back to the first level of recursion. This level's first recursive call is finished so we recurse right (12). We are not at a base case so we recurse left (13).

Merge Sort Complete



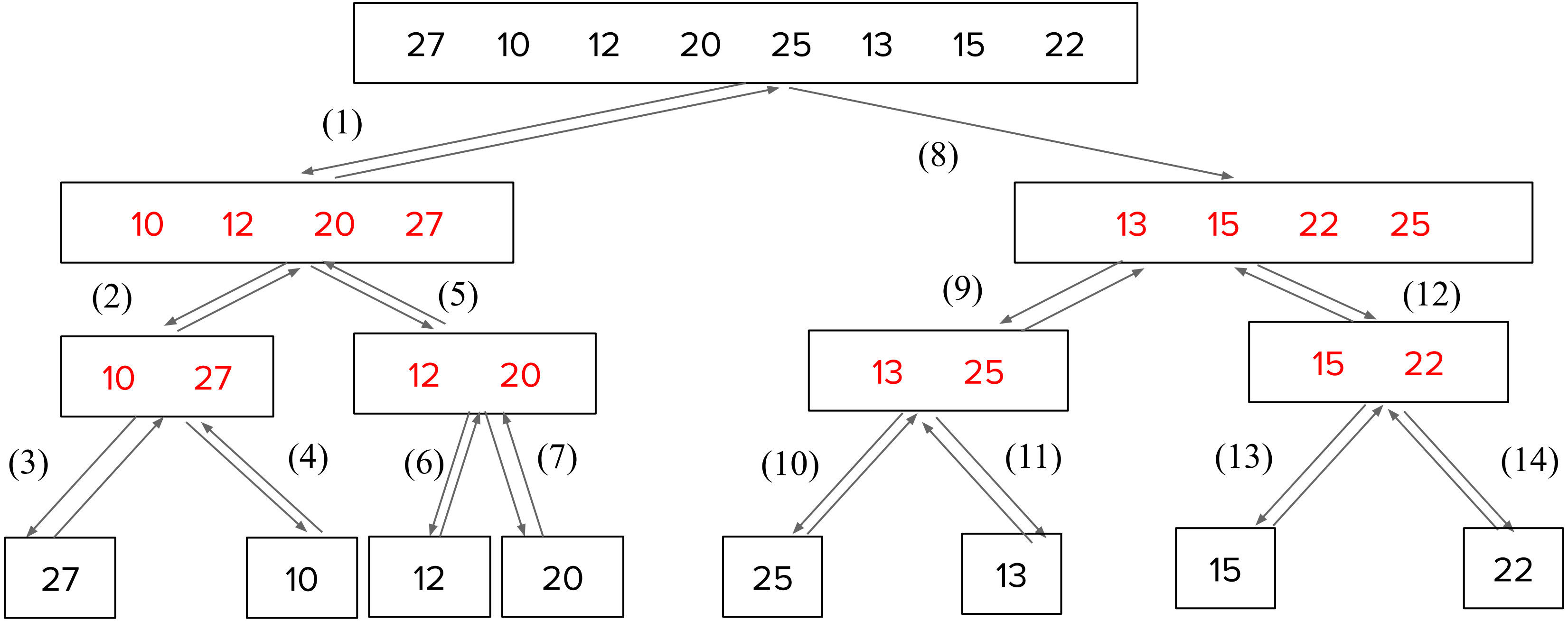
- We pop back to the second level of recursion and recurse right.
- We immediately reach another base case and pop back up to the second level.

Merge Sort Complete



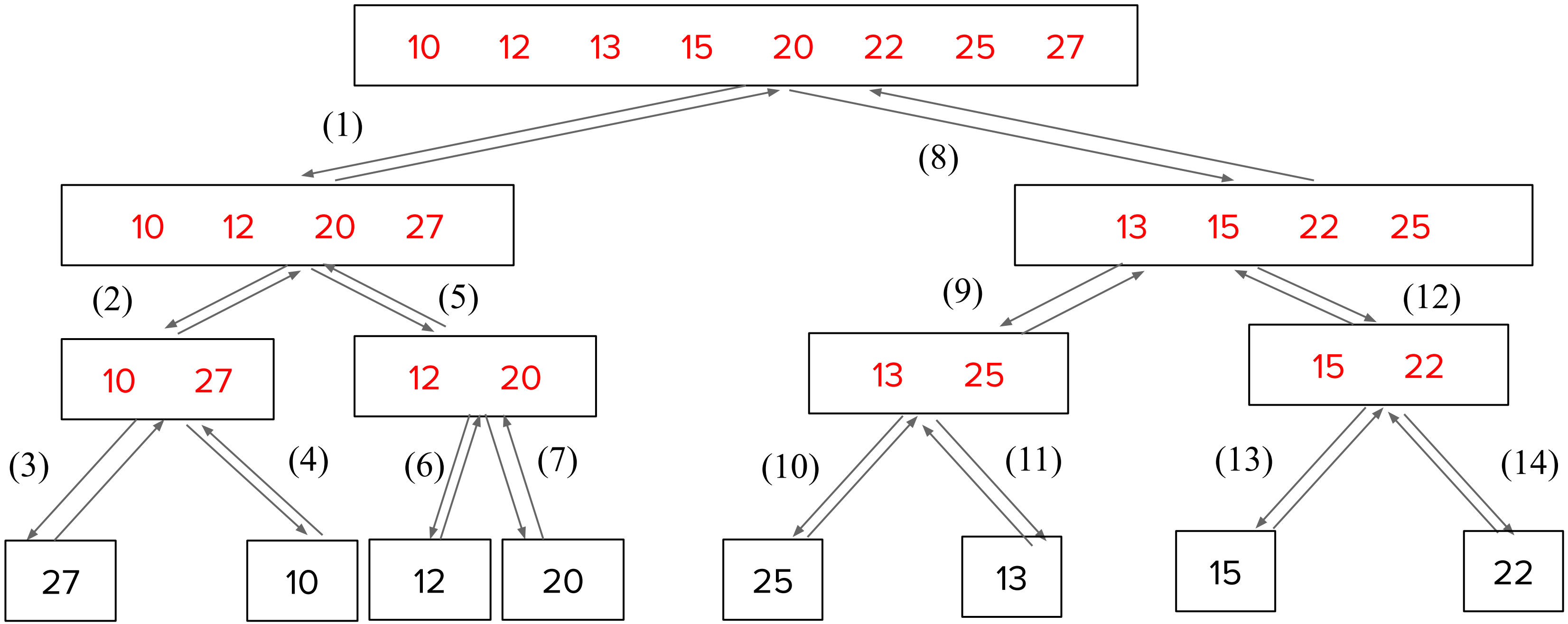
- The second level of recursion has completed both of its recursive calls.
- Merge is called on U and V and the results are placed in the second level's S array.

Merge Sort Complete



- The first level of recursion has completed both of its recursive calls.
- Merge is called on U and V and the results are placed in the first level's S array.

Merge Sort Complete



- The top level has completed both of its recursive calls.
- Merge is called on U and V and the results are placed in the top level's S array.

Merge

- The `merge` procedure is the key component to `MergeSort`.
- When $n = 8$, in the final call to `merge`, we have two sorted sublists U and V. We iterate through both, placing the values in the correct order in S

k	U	V	S(Result)
1	**10** 12 20 27	**13** 15 22 25	10
2	10 **12** 20 27	**13** 15 22 25	10 12
3	10 12 **20** 27	**13** 15 22 25	10 12 13
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25
—	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 ← Final values

*Items compared are in boldface.

Merge

```
void merge(int uLen, int vLen, const keytype V[], const keytype U[], keytype S[])
     index i, j, k = 1;
     while(i <= uLen && j <= vLen)
          if (U[i] < V[j])
               S[k] = U[i];
               i++;      // i tracks how many values from U have been placed in S
          else
               S[k] = V[j];
               j++;      // j tracks how many values from V have been placed in S
          k++;           // k tracks how many total values have been placed in S
     if (i > uLen)
          copy V[j] through V[vLen] to S[k] through S[uLen+vLen];
     else
          copy U[i] through U[uLen] to S[k] through S[uLen+vLen];
```

- After the loop, if `i > uLen`, we have placed every item from *U* into *S*. All remaining values in *V* are placed in *S*.

Merge Sort Analysis

```
void mergeSort(int n, keytype S[])
     if (n > 1)
          const int uLen = ⌊n/2⌋, vLen = n - uLen;
          keytype U[1..uLen], V[1..vLen];
          copy S[1] through S[uLen] to U[1] through U[uLen]
          copy S[uLen+1] through S[n] to V[1] through V[vLen]
          mergeSort(uLen, U)
          mergeSort(vLen, V)
          merge(uLen, vLen, U, V, S)
```

- What is the basic operation?

Merge Sort Analysis

```
void mergeSort(int n, keytype S[])
     if (n > 1)
          const int uLen = ⌊n/2⌋, vLen = n - uLen;
          keytype U[1..uLen], V[1..vLen];
          copy S[1] through S[uLen] to U[1] through U[uLen]
          copy S[uLen+1] through S[n] to V[1] through V[vLen]
          mergeSort(uLen, U)
          mergeSort(vLen, V)
          merge(uLen, vLen, U, V, S)
```

- What is the basic operation?
  - `MergeSort`'s basic operation occurs in the `merge` procedure.
  - This operation occurs once at the top level

Merge Analysis

```
void merge(int uLen, int vLen, const keytype V[],
      const keytype U[], keytype S[])
      index i, j, k = 1;
      while(i <= uLen && j <= vLen)
            if (U[i] < V[j])
                  S[k] = U[i];
                  i++;
            else
                  S[k] = V[j];
                  j++;
            k++
      if (i > uLen)
            copy V[j] through V[vLen] to S[k] through S[uLen+vLen];
      else
            copy U[i] through U[uLen] to S[k] through S[uLen+vLen];
```

Before analyzing `MergeSort` as a whole, we need to analyze `Merge`.

- **Basic Operation**: ?

Merge Analysis

Before analyzing `MergeSort` as a whole, we need to analyze `Merge`.

- **Basic Operation**: The comparison of `U[i]` with `V[j]`
- **Input Size**: `uLen` and `vLen`: the # of items in U and V

Does `Merge` have an every-case time complexity?

Merge Analysis

Before analyzing `MergeSort` as a whole, we need to analyze `Merge`.

- **Basic Operation**: The comparison of `U[i]` with `V[j]`
- **Input Size**: `uLen` and `vLen`: the # of items in *U* and *V*

Does `Merge` have an every-case time complexity? **No!**

```
while(i <= uLen && j <= vLen)
```

➢ We break out of the loop early once $i =$ `uLen` or $j =$ `vLen`. (All elements from *U* or *V* have been placed in *S*)

Worst case scenario?

Merge Analysis

Before analyzing `MergeSort` as a whole, we need to analyze `Merge`.

- **Basic Operation**: The comparison of `U[i]` with `V[j]`
- **Input Size**: `uLen` and `vLen`: the # of items in U and V

Does `Merge` have an every-case time complexity? **No!**

```
while(i <= uLen && j <= vLen)
```

➢ We break out of the loop early once i = `uLen` or j = `vLen`. (All elements from U or V have been placed in S)

Worst case scenario? Every item in both arrays has to be merged.

- If each item from U is placed in S and all but 1 from V is placed in S before we break from the loop, the max # of iterations have taken place.

Therefore, $W(uLen, vLen) = uLen + vLen - 1$

MergeSort Analysis

- Now that `Merge` has been analyzed, we analyze `MergeSort` as a whole.
- **Basic Operation**: # of comparisons that take place when `merge` is called in both recursive calls to `MergeSort` and the top-level `merge` call:

```
mergeSort(uLen, U)
mergeSort(vLen, V)
merge(uLen, vLen, U, V, S)
```

- In the top-level of `MergeSort`, `merge` is called once. We have already determined the worst case for a single `merge`: `uLen + vLen - 1`
- Therefore, we have: $W_n = W_{uLen} + W_{vLen} + (uLen + vLen - 1)$
  - W_{uLen}: # of operations when calling `mergeSort` on U
  - W_{vLen}: # of operations when calling `mergeSort` on V
  - $(uLen + vLen - 1)$: # of operations at the top level

MergeSort Analysis

$W_n = W_{uLen} + W_{vLen} + (uLen + vLen - 1)$

- We can assume n is a power of 2. In that case $uLen = vLen = n / 2$
- $uLen$ and $vLen$ are exactly half of n. Therefore:
  - $uLen + vLen = n$, which we can substitute into our equation
  - We end up with:
    - $W_{n/2} + W_{n/2} + n - 1$

  $= \quad 2W_{n/2} + n - 1$

We end up with:

$$W_n = 2W_{n/2} + n - 1 \quad \text{for } n > 1, n \text{ a power of } 2$$

MergeSort Analysis

```
void mergeSort(int n, keytype S[])
     if (n > 1)
          const int uLen = ⌊n/2⌋, vLen = n - uLen;
          keytype U[1..uLen], V[1..vLen];
          copy S[1] through S[uLen] to U[1] through U[uLen]
          copy S[uLen+1] through S[n] to V[1] through V[vLen]
          mergeSort(uLen, U)
          mergeSort(vLen, V)
          merge(uLen, vLen, U, V, S)
```

So far we have:

$$W_n = 2W_{n/2} + n - 1 \quad \text{for } n > 1,\ n \text{ a power of } 2$$

What else do we need for this recurrence relation?

MergeSort Analysis

```
void mergeSort(int n, keytype S[])
     if (n > 1)
          const int uLen = ⌊n/2⌋, vLen = n - uLen;
          keytype U[1..uLen], V[1..vLen];
          copy S[1] through S[uLen] to U[1] through U[uLen]
          copy S[uLen+1] through S[n] to V[1] through V[vLen]
          mergeSort(uLen, U)
          mergeSort(vLen, V)
          merge(uLen, vLen, U, V, S)
```

So far we have:

$$W_n = 2W_{n/2} + n - 1 \quad \text{for } n > 1, n \text{ a power of } 2$$

What else do we need for this recurrence relation?

➢ We do **no** operations at the base case when $n = 1$: $W_1 = 0$

MergeSort Analysis

$W_n = 2W_{n/2} + n - 1$ for $n > 1$, n a power of 2
$W_1 = 0$

- **First step**:

MergeSort Analysis

$W_n = 2W_{n/2} + n - 1$ for $n > 1$, n a power of 2
$W_1 = 0$

- **First step**: Calculate $W_{n/2}$
  - $W_{n/2} = 2W_{n/4} + (n / 2) - 1$
- **Second step**: Plug the result in to W_n:
  - $W_n = 2[\mathbf{2W_{n/4} + (n / 2) - 1}] + n - 1$
  - $= 4W_{n/4} + 2n - 3$

MergeSort Analysis

$4W_{n/4} + 2n - 3$

- Calculate $W_{n/4}$:

  $W_{n/4} = 2W_{n/8} + (n/4) - 1$

  Plug it in:

  - $4[\mathbf{2W_{n/8} + (n/4) - 1}] + 2n - 3$
  - $8W_{n/8} + 4(n/4) - 4 + 2n - 3$

What is the pattern? (we are on the 3rd level of recursion)

MergeSort Analysis

$8W_{n/8} + 4(n/4) - 4 + 2n - 3 \quad =$

$2^3W_{n/8} + n - 4 + 2n - 3 \quad =$

$2^3W_{n/8} + 3n - 7 \quad =$

$2^3W_{n/8} + 3n - (2^3 - 1) \quad =$

$2^kW_{n/2^k} + kn - (2^k - 1)$

Note that we turned 7 into $2^3 - 1$ (i.e. 8 - 1).

In the 2nd level of recursion we had -3 (i.e. $2^2 - 1$) and in the first level we had -1 (i.e. $2^1 - 1$)

MergeSort Analysis

$2^k W_{n/2^k} + kn - (2^k - 1)$

We reach the base case and stop recursion at W_1, which returns 0.

$n = 2^k$ at this point. This leaves us with:

➢ $2^k \times W_1 + kn - (2^k - 1)$

Since $n = 2^k$, we can also say that $k = \lg n$.

Therefore:

- $2^{\lg n} \times 0 + n\lg n - (2^{\lg n} - 1) = n\lg n - n + 1$
- `MergeSort`'s worst case is **O(*n*lg*n*)**

MergeSort Analysis

- We covered an exhaustive proof of `MergeSort`'s order.
- Often we simply want to intuitively know what the order of an algorithm is.

- Any time we split data of size n in half recursively, it takes lg$n$ time.
- However, with `MergeSort`, we have to iterate through all the data each recursive call in the worst case, so we multiply lg$n$ by n, leaving us with $n$lg$n$

MergeSort

```
void mergeSort(int n, keytype S[])
     if (n > 1)
          const int uLen = ⌊n/2⌋, vLen = n - uLen;
          keytype U[1..uLen], V[1..vLen];
          copy S[1] through S[uLen] to U[1] through U[uLen]
          copy S[uLen+1] through S[n] to V[1] through V[vLen]
          mergeSort(uLen, U)
          mergeSort(vLen, V)
          merge(uLen, vLen, U, V, S)
```

- When analyzing algorithms, we usually consider their time complexity. However, it is also important to be aware of an an algorithm’s **memory complexity**.
- Is there anything about the given `MergeSort` algorithm that uses more memory than it needs?

MergeSort

```
void mergeSort(int n, keytype S[])
     if (n > 1)
          const int uLen = ⌊n/2⌋, vLen = n - uLen;
          keytype U[1..uLen], V[1..vLen];
          copy S[1] through S[uLen] to U[1] through U[uLen]
          copy S[uLen+1] through S[n] to V[1] through V[vLen]
          mergeSort(uLen, U)
          mergeSort(vLen, V)
          merge(uLen, vLen, U, V, S)
```

- When analyzing algorithms, we usually consider their time complexity. However, it is also important to be aware of an an algorithm's **memory complexity**.
- Is there anything about the given `MergeSort` algorithm that uses more memory than it needs?
  - In each call to `MergeSort`, we create two new arrays.
  - With a large enough starting array, this can take considerable memory.

MergeSort in-place

- The version of `MergeSort` we covered helps clearly demonstrate how the algorithm works, but it wastes space: each call to `MergeSort` creates two new arrays, U and V
- Typically, we want to do an **in-place** sort
- An in-place sort does not use any more space than is required to perform the sorting. i.e. We do not create new arrays U and V to pass to recursive calls to `MergeSort`, but rather pass `MergeSort` two integer values indicating indices within S.
- These integers tell `MergeSort` which part of the original array to sort.
- All sorting takes place within the original array S.

MergeSort in-place

```
void mergesort2 (index low, index high)
{
      index mid;
      if (low < high)
      {
            mid = ⌊(low + high) / 2⌋;
            mergesort2(low, mid);
            mergesort2(mid + 1, high);
            merge2(low, mid, high);
      }
}
```

The first call to `mergeSort2` passes 1 for `low` and n for `high`.

Merge in-place

```
void merge2 (index low, index mid, index high)
     index i = low, j = mid + 1, k = low;
     keytype U[low...high];                    // local array needed for merging
     while (i <= mid and j <= high)
          if (S[i] < S[j])
               U[k] = S[i];
               i++;
          else
               U[k] = S[j];
               j++;
          k++;
     if (i > mid)
          move S[j] through S[high] to U[k] through U[high]
     else
          move S[i] through S[mid] to U[k] through U[high];
     move U[low] through U[high] to S[low] through S[high];
```

In-Class Exercise

1. Draw the recursive tree created when sorting the following array using `MergeSort`. Number the steps!

   { 123, 34, 189, 56, 150, 12, 9, 240 }

2. Sort 65, 60_1, 60_2, 60_3 in nondecreasing order using `MergeSort`. A sorting algorithm is called **stable** if it preserves the relative order of any two equal elements in its input. Is `MergeSort` stable? (hint: consider how the Merge pseudocode from the slides works)