# Merge Sort

Brian E. Lavender

April 15, 2016

## Merge Sort Concept

Sorts an array of values

- **Divide and Conquer**:
- **Recursive Routine**
- **Complexity**. O($N\log_2 N$)

- **Disadvantage**: auxiliary array

## Separate video

Harvard University Merge Sort video

## Merge two sorted halves Example

Two sorted arrays of size *k/2*.
**Merge** them to destination of size *k*.

min(4,8)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| **4** | 15 | 16 | 50 |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| **8** | 23 | 42 | 108 |

| 0 | 1 | 2 | 3 |
|---|---|---|---|

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| _ | _ | _ | _ | _ | _ | _ | _ |

## Merge Half Arrays To Destination (Step 1)

Move *4* from left_half to destination

min(15,8)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| * | **15** | 16 | 50 |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| **8** | 23 | 42 | 108 |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | | | | | | | |

## Merge Half Arrays To Destination (Step 2)

Move 8 from right_half to destination.

min(15,23)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | **15** | 16 | 50 |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| * | **23** | 42 | 108 |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 8 | | | | | | |

## Merge Half Arrays To Destination (Step 3)

Move 15 from left half to destination.

min(16,23)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | * | **16** | 50 |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | **23** | 42 | 108 |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 15 | | | | | |

## Merge Half Arrays To Destination (Step 4)

Move 16 from left_half to destination.

min(50,23)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | * | **50** |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | **23** | 42 | 108 |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 15 | *16* | | | | |

## Merge Half Arrays To Destination (Step 5)

Move 23 from the right_half to destination.

min(50,42)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | | **50** |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | * | **42** | 108 |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 15 | 16 | *23* | | | |

## Merge Half Arrays To Destination (Step 6)

Move 42 from the right_half to destination.

min(50,108)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | | **50** |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | * | **108** |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 15 | 16 | 23 | *42* | | |

## Merge Half Arrays To Destination (Step 7)

Move 50 from the left_half to destination.

empty(left_half)?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | | * |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | | **108** |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 15 | 16 | 23 | 42 | *50* | |

## Merge Half Arrays To Destination (Step 8)

Move 108 from the right_half to destination.

(have_items(left_array) or have_items(right_array))?

Left Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | | |

Right Half (size 4)

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| | | | * |

Destination (size 8)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 15 | 16 | 23 | 42 | 50 | *108* |

Destination array **sorted**

8 steps *linear time*.

## The C++ merge code

C++ Code illustrating the above merge. Consider four cases.

1. Either *left_half* or *right_half* have elements to move to the destination.
2. All elements from *left_half* moved to destination. Move minimum element from *right_half*.
3. All elements from *right_half* moved to destination. Move minimum element from *left_half*.
4. Move smaller element from *left_half* or *right_half* to destination.

```
const int SIZE_HALF_LEFT = 4;
const int SIZE_HALF_RIGHT = 4;
const int SIZE_DEST = 8;

int left_half[SIZE_HALF_LEFT] = {4, 15, 16, 50};
int right_half[SIZE_HALF_RIGHT] = {8, 23, 42, 108};
int destination[SIZE_DEST];

// index counters
// i - left array , j - right array, k - destination array
i = 0; j = 0; k = 0;

// Merge the two half lists together
// while we have elements in either of the two lists
while (i < SIZE_HALF_LEFT || j < SIZE_HALF_RIGHT) {
  // Completed the first half
  if  ( i >= SIZE_HALF_LEFT ) {
    destination[k] = right_half[j];
    j++;
  }

  //  Completed the second half
  else if (j >= SIZE_HALF_RIGHT) {
    destination[k] = left_half[i];
    i++;
  }

  // pick the smallest element from one
  // of the two lists
  else if (left_half[i] < right_half[j]) {
    destination[k] = left_half[i];
    i++;
  } else {
    destination[k] = right_half[j];
    j++;
  }

  // increment our counter for destination
  k++;
```

```
40 }
```

## Next Challenge

Merge assumes *left_half* and *right_half* **already sorted**.

If we start with the following.

```
int destination[] = {108, 15, 50, 4, 8, 42, 23, 16};
```

Halves **not sorted**!

```
int left_half[] = {108, 15, 50, 4};
int right_half[] = {8, 42, 23, 16};
```

Keep dividing in half?

## Get to the bottom of this

Repeatedly divide in half!

$\log_2 8 = 3$ levels

```
---       8
^         4           4
3 levels  2     2     2     2
v         1  1  1  1  1  1  1  1
---
```

## Divide array into halves

Repeatedly call merge_sort on halves!

```
int destination[] = {108, 15, 50, 4, 8, 42, 23, 16};
```

### First Division

Sorted? No

```
int left_half[] = {108, 15, 50, 4};
int right_half[] = {8, 42, 23, 16};
```

### Second Division

Sorted? No

```
int left_half[] = {108, 15};
int right_half[] = {50, 4};

int left_half[] = {8, 42};
int right_half[] = {23, 16};
```

### Third Division

Sorted? **Yes**

```
int left_half[] = {108};
int right_half[] = {15};

int left_half[] = {50};
int right_half[] = {4};

int left_half[] = {8};
int right_half[] = {42};

int right_half[] = {23};
int right_half[] = {16};
```

## Merge two half arrays of size 1 to destination of size 2

Build it back up

min(23,16)?

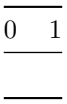
Left Half (size 1)

| |
|---|
| 0 |
| **23** |

Right Half (size 1)

| |
|---|
| 0 |
| **16** |

Destination (size 2)

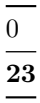



## Merge two half arrays of size 1 to destination of size 2 (step 1)

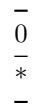
Move *16* from right_half to destination

empty(right_half)?

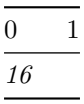
Left Half (size 1)




Right Half (size 1)



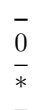

Destination (size 2)



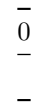

## Merge two half arrays of size 1 to destination of size 2 (step 2)

Move 23 from first_half to destination

Left Half (size 1)




Right Half (size 1)




Destination (size 2)

| 0 | 1 |
|---|---|
| 16 | 23 |

We merged the simplest two half arrays of size 1 back into a dest array of 2 items.

## Algorithm for recursion

1. Figure out our base case
2. Determine our recursive step.

## Base Case

1 item already sorted.

```
void merge_sort(int destination[], int dest_size) {
    // base case. Array size is one.
    if (dest_size == 1)
        return;

}
```

## Recursive Step

1. Make new left_half and right_half. Copy data from destination to halves.
2. Call *merge_sort* on each half array.
3. Merge the sorted halves together.
4. Return the destination array

## Divide the Array

Making new halves.

```
void merge_sort(int destination[], int dest_size) {
  int left_size, right_size;
  if (dest_size == 1)
    return;
  else {
    // ***********
    // Make two new half arrays
    // ***********
    // Make two new arrays: left_array half and right_array half
    //  Integer division for size: 5 / 2 -> 2
    size_left = dest_size  / 2 ;
    size_right = array_size - size_left;
    int left_array[size_left];
    int right_array[size_right];
    for (i=0; i< size_left; i++)
      left_array[i] = x[i];
    for (i=0; i< size_right; i++)
      right_array[i] = x[size_left + i];

    // call merge_sort on left_array
    // call merge_sort on right_array
    // merge the left_array and right_array together
  }
  // return our sorted array
  return;
}
```

## Recursively Call Merge sort on halves

Recursively call *merge_sort* on the two halves.

```
void merge_sort(int destination[], int dest_size) {
  int left_size, right_size;
  if (size == 1)
    return;
  else {
    // Make two new arrays: left_array half and right_array half
    //  Integer division for size: 5 / 2 -> 2
    size_left = dest_size  / 2 ;
    size_right = dest_size - size_left;
    int left_array[size_left];
    int right_array[size_right];
    for (i=0; i< size_left; i++)
      left_array[i] = x[i];
```

```
    for (i=0; i< size_right; i++)
      right_array[i] = x[size_left + i];

    // ***************
    // Recursively Call Merge sort on halves
    // ***************
    merge_sort(left_array,size_left);
    merge_sort(right_array,size_right);

    // Merge the two halves
    // Our inductive step
  }
  // return our sorted array
  return;
}
```

## Side Step

Pause for a moment and make a function out our merge example previously illustrated.

```
void merge(int left[], int right[], int destination[],
       int size_of_left_half, int size_of_right_half) {
  int i = 0; // works on left half
  int j = 0; // works on right half
  // The merge
  while (i < size_of_left_half || j < size_of_right_half) {
    // Completed the first half
    if  ( i >= size_of_left_half ) {
      destination[k] = right[j];
      j++;
    }
    //  Completed the second half
    else if (j >= size_of_right_half) {
      destination[k] = left[i];
      i++;
    }
    // pick the smallest one
    else if (left_half[i] < right_half[j]) {
      destination[k] = left[i];
      i++;
    } else {
      destination[k] = right[j];
      j++;
```

```
    }
    k++;
  }
}
```

## Merge the two halves

Merge left_half and right_half back to destination.

```
void merge_sort(int destination[], int dest_size) {
  int left_size, right_size;
  if (dest_size == 1)
    return;
  else {
    // Make left_array half and right_array half
    size_left = dest_size  / 2 ;
    size_right = dest_size - size_left;
    int left_array[size_left];
    int right_array[size_right];
    for (i=0; i< size_left; i++)
      left_array[i] = destination[i];
    for (i=0; i< size_right; i++)
      right_array[i] = destination[size_left + i];

    // recursively call merge_sort on the
    // left_half and right_half
    merge_sort(left_array,size_left);
    merge_sort(right_array,size_right);

    // **************
    // Merge the two sorted halves
    // Our inductive step
    // **************
    merge(left_array,right_array,destination,size_left,size_right);
  }
  // return our sorted array
  return;
}
```

## Return

Who called me?

## Final Code

The following is the entire merge sort program with a main driver. It also illustrates an array of an odd number of elements.

```
#include <iostream>

using namespace std;

void display(int x[], int size) {
  int i;
  cout << "Size is " << size << endl;
  for (i=0; i < size; i++)
    cout  << x[i] << " ";
  cout << endl;
}

void merge(int left[], int right[], int destination[],
           int size_of_left_half, int size_of_right_half) {
  int i = 0; // works on left half
  int j = 0; // works on right half
  int k = 0; // merged array count
  // The merge
  while (i < size_of_left_half || j < size_of_right_half) {
    // Completed the first half
    if  ( i >= size_of_left_half ) {
      destination[k] = right[j];
      j++;
    }
    //  Completed the second half
    else if (j >= size_of_right_half) {
      destination[k] = left[i];
      i++;
    }
    // pick the smallest one
    else if (left[i] < right[j]) {
      destination[k] = left[i];
      i++;
    } else {
      destination[k] = right[j];
      j++;
    }
    k++;
  }

}
```

```
42
43  // precondition
44  // size of the array is 1 or greater
45  void merge_sort(int destination[], int dest_size) {
46    int size_left = 0, size_right = 0;
47    int i;
48
49    // Our base case.
50    // An array of one element is considered sorted
51    if (dest_size == 1)
52      return ;
53
54    // recursively call merge_sort of two halves.
55    else {
56      size_left = dest_size  / 2 ;
57      size_right = dest_size - size_left;
58
59      // temporary arrays for divide portion
60      int left_array[size_left];
61      int right_array[size_left];
62
63      // Copy data into new temp array for the left.
64      for (i=0; i< size_left; i++)
65        left_array[i] = destination[i];
66
67      // Copy data into new temp array for the right.
68      for (i=0; i< size_right; i++)
69        right_array[i] = destination[size_left + i];
70
71      // recursively call merge_sort on the
72      // left_half and right_half
73      merge_sort(left_array,size_left);
74      merge_sort(right_array,size_right);
75
76      // Merge the two halves
77      // Our inductive step
78      merge(left_array,right_array,destination,size_left,size_right);
79    }
80
81    return ;
82  }
83
84
85  int main()
86  {
87
```

```
88    const int SIZE_DEST = 11;
89
90    int destination[SIZE_DEST] = {101,99,21,12,11,25,20,4,2,17,3};
91
92    cout << "Before sort" << endl;
93    display(destination, SIZE_DEST);
94
95    merge_sort(destination, SIZE_DEST);
96
97    cout << "After sort" << endl;
98    display(destination, SIZE_DEST);
99
100   return 0;
101 }
```

## Questions?!

## Exercise

Randomly arrange the cups and see if you can perform the merge sort.

## References


1. Dale, Nell, *C++ Plus Data Structures, 3rd Ed.*, Jones and Bartlett Publishers, 2003, pp 601-608.