COP 3502: Computer Science I
Spring 2004

– Note Set 11 –
Searching and Sorting – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

Bubble Sort

- The general idea with the bubble sort is to allow the smaller elements to move toward the "top" (i.e. front) of the list while larger elements move toward the "bottom" (i.e., rear) of the list.
- There are several different variants of the bubble sort, we'll look only at one of them as they all have the same asymptotic behavior.

Bubble Sort Algorithm

```
BubbleSort (list, n)
    // list       the elements to be sort
    // n          the number of elements in the list

numberofPairs = n;
swappedElements = true;
while swappedElements do
      numberofPairs = numberofPairs – 1;
      swappedElements = false;
      for (i = ; i <= numberofPairs; i++){
           if ( list[i] > list[i+1] ){
                swap( list[i], list[i+1]);
                swappedElements = true;
           }
      endfor;
endwhile;
end.
```

Bubble Sort Algorithm

- The bubble sort algorithm makes a number of passes through the list of elements. On each pass it compares adjacent element values. If they are out of order, they're positions in the list are reversed via a swap.
- Each pass starts at the beginning of the list and compares the elements in locations 1 and 2, then the elements in locations 2 and 3, then locations 3 and 4 and so on, swapping any that are out of order.
- On the first pass, once the largest element is encountered, it will be swapped with all of the remaining elements, moving it to the end (bottom) of the list.
- The second pass, therefore, no longer needs to look at the last element in the list. The second pass will move the second largest element in the list until it is in the next to last location.

Bubble Sort Algorithm (cont.)

- This process repeats with each additional pass moving one more of the larger elements down the list.
- During all of this, the smaller elements are migrating toward the front (top) of the list.
- If on any pass, no swaps occur, then all of the elements are now in order and the algorithm can terminate.
  - There are some versions of the bubble sort that will not make this check and will therefore continue to make passes through the list until all possible passes have completed. This is clearly inefficient, but be aware that such implementations of this algorithm do exist.
- Notice that each pass has the potential for moving a number of elements closer to their final positions, even though only the largest element on that pass is guaranteed to wind up in its final location.

Bubble Sort Example

initial unsorted list

34	15	7	3	56	22	45	9	12
15	7	3	34	22	45	9	12	56
7	3	15	22	34	9	12	45	56
3	7	15	22	9	12	34	45	56
3	7	15	9	12	22	34	45	56
3	7	9	12	15	22	34	45	56

Notice that on the fifth pass that no swaps occurred, so the algorithm terminated with a sorted list after only 5 passes.

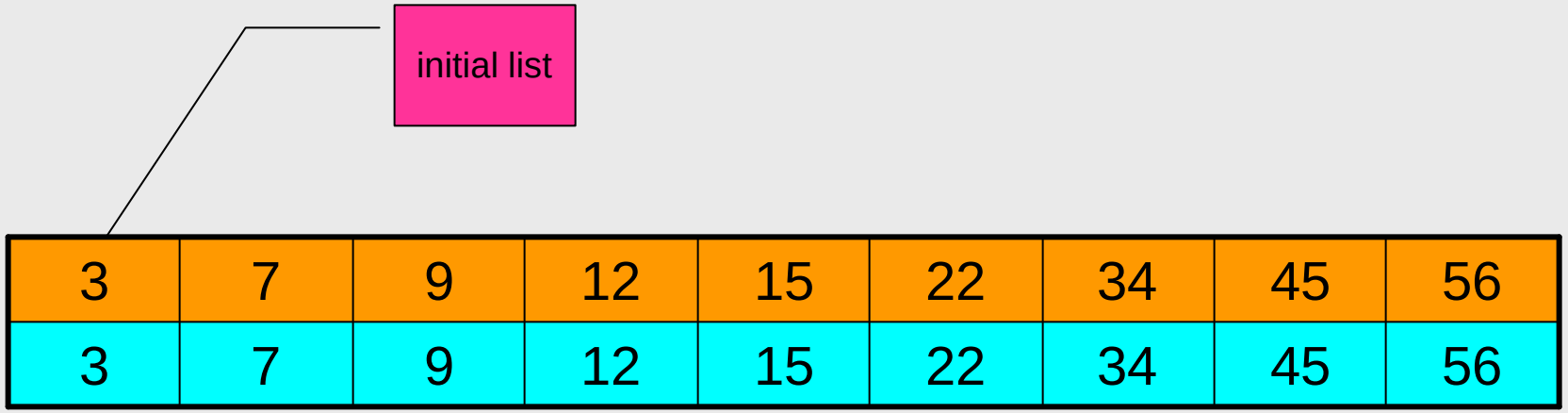
Sorted part of the list is shaded.

Bubble Sort Algorithm Analysis (cont.)

Best Case

- Don't let the swappedElements flag give you the wrong impression how this algorithm works. Consider the situation in which this algorithm will do the least amount of work.
- On the first pass, the for loop must fully execute, so at least n-1 comparisons will be done. Two possibilities must be considered: there is at least 1 swap or there are no swaps.
- In the first case, the swappedElements flag will be set to true, which will cause the while loop to execute a second time, which will do another n-2 comparisons.
- In the second case, because there are no swaps, the swappedElements flag will still be false and the algorithm will end.
- So the best case will be n-1 comparisons, which is O(n) and this will occur when the input data is already in sorted order.

Bubble Sort Example – Best Case



Sorted part of the list is shaded.

Bubble Sort Example – Worst Case

56	45	34	22	15	12	9	7	3
45	34	22	15	12	9	7	3	56
34	22	15	12	9	7	3	45	56
22	15	12	9	7	3	34	45	56
15	12	9	7	3	22	34	45	56
12	9	7	3	15	22	34	45	56
9	7	3	12	15	22	34	45	56
7	3	9	12	15	22	34	45	56
3	7	9	12	15	22	34	45	56

Sorted part of the list is shaded.

Bubble Sort Algorithm Analysis (cont.)

Worst Case

- If the best case occurs when the input data is already in sorted order, we might want to see if having the input data in reverse sorted order will lead to the worst case performance of the algorithm.
- If the largest element is first, it will be swapped with every other element down the list until it is in the last position.
- At the start of the first pass the second largest element in the list was in the second position but it was swapped into the first position when it was compared with the largest element that was in the first position. Therefore, at the start of the second pass, the largest remaining element is again in the first position of the list, and it will be swapped with every other element in the list until it is in the next to last position.
- This will be repeated for every other element in the list.

Bubble Sort Algorithm Analysis (cont.)

Worst Case (cont.)

- Thus, the algorithm must execute the for loop a total of n-1 times, which means that we guessed correctly that the input data being in reverse sorted order does indeed deliver worst case performance for this algorithm.
- How many comparisons are done in the worst case?
  - The first pass will do n-1 comparisons on adjacent elements.
  - The second pass will do n-2 comparisons on adjacent elements.
  - Each successive pass will reduce the number of comparisons by 1, which will leads to the following analysis:

$$\text{worst}(n) = \sum_{i=n-1}^{1} i = \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2} \approx \frac{1}{2}n^2 = O\left(n^2\right)$$

Bubble Sort Algorithm Analysis (cont.)

Average Case

- As we saw for the worst case, there are n-1 repetitions of the inner for loop. For the average case, we'll assume that it is equally likely that on any of the passes there will be no swaps done.
- We need to determine the number of comparisons are done in each of these possibilities.
  - If we stop after 1 pass, a total of n-1 comparisons have been made.
  - If we stop after 2 passes, a total of n-1 + n-2 comparisons have been made.
- Lets' say that C(i) will calculate how many comparisons are done on the first *i* passes.
- Since the algorithm terminates when there are no swaps done, the average case is determined by looking at all of the places the bubble sort could stop. This is given by:

Bubble Sort Algorithm Analysis (cont.)

Average Case (cont.)

$$\text{average}(n) = \frac{1}{n-1}\sum_{i=1}^{n-1} C(i)$$

$$\text{where } C(i) = \sum_{j=n-1}^{i} j = \sum_{j=i}^{n-1} j = \sum_{j=1}^{n-1} j - \sum_{j=1}^{i=1} j = \frac{(n-1)n}{2} - \frac{(i-1)i}{2} = \frac{n^2 - n - i^2 + i}{2}$$

Substituting we have: $\text{average}(n) = \frac{1}{n-1}\sum_{i=1}^{n-1}\left(\frac{n^2 - n - i^2 + i}{2}\right)$

Note that *n* is a constant relative to *i*, so we have:

$$\text{average}(n) = \frac{1}{n-1}\left[(n-1)\times\frac{n^2-n}{2} + \sum_{i=1}^{n-1}\left(\frac{-i^2+i}{2}\right)\right] = \frac{n^2-n}{2} + \frac{1}{2(n-1)}\left[\sum_{i=1}^{n-1} -i^2 + \sum_{i=1}^{n-1} i\right]$$

Bubble Sort Algorithm Analysis (cont.)

Average Case (cont.)

$$\text{average}(n) = \frac{1}{n-1}\left[(n-1)\times\frac{n^2-n}{2} + \sum_{i=1}^{n-1}\left(\frac{-i^2+i}{2}\right)\right] = \frac{n^2-n}{2} + \frac{1}{2(n-1)}\left[\sum_{i=1}^{n-1} -i^2 + \sum_{i=1}^{n-1} i\right]$$

Solving we have:

$$\text{average}(n) = \frac{n^2-n}{2} + \frac{1}{2(n-1)}\left[\frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2}\right] = \frac{n^2-n}{2} - \frac{n(2n-1)}{12} + \frac{n}{4}$$

Reducing gives:

$$\text{average}(n) = \frac{6n^2-6n-2n^2+n+3n}{12} = \frac{4n^2-2n}{12} \approx \frac{1}{3}n^2 = O(n^2)$$

Summary of Bubble Sort

- "Smart" versions of the algorithm terminate whenever no swaps occur on a given pass.
- Analysis is the same whether bubbling smallest element to front or largest element to rear.

Bubble Sort	
best case	O(n)
average case	$O(n^2)$
worst case	$O(n^2)$

Merge Sort

- Merge sort is a recursive sorting algorithm. It is based on the idea that merging two sorted lists can be done in linear time.
- Since a list that contains only one item is by definition sorted, merge sort will break a list down into one-element pieces and then sort as it merges the pieces back together.
- All of the "work" for merge sort occurs during the merging of the two lists.

MergeSort Algorithm

```
MergeSort (list, first, last)
    // list        the elements to be sorted
    // first       the index of the first element in the list
    // last        the index of the last element in the list

   if ( first < last ){
        middle = (first + last) / 2;
        MergeSort(list, first, middle);
        MergeSort(list, middle+1, last);
        MergeLists(list, first, middle, middle+1, last);
    }
end.
```

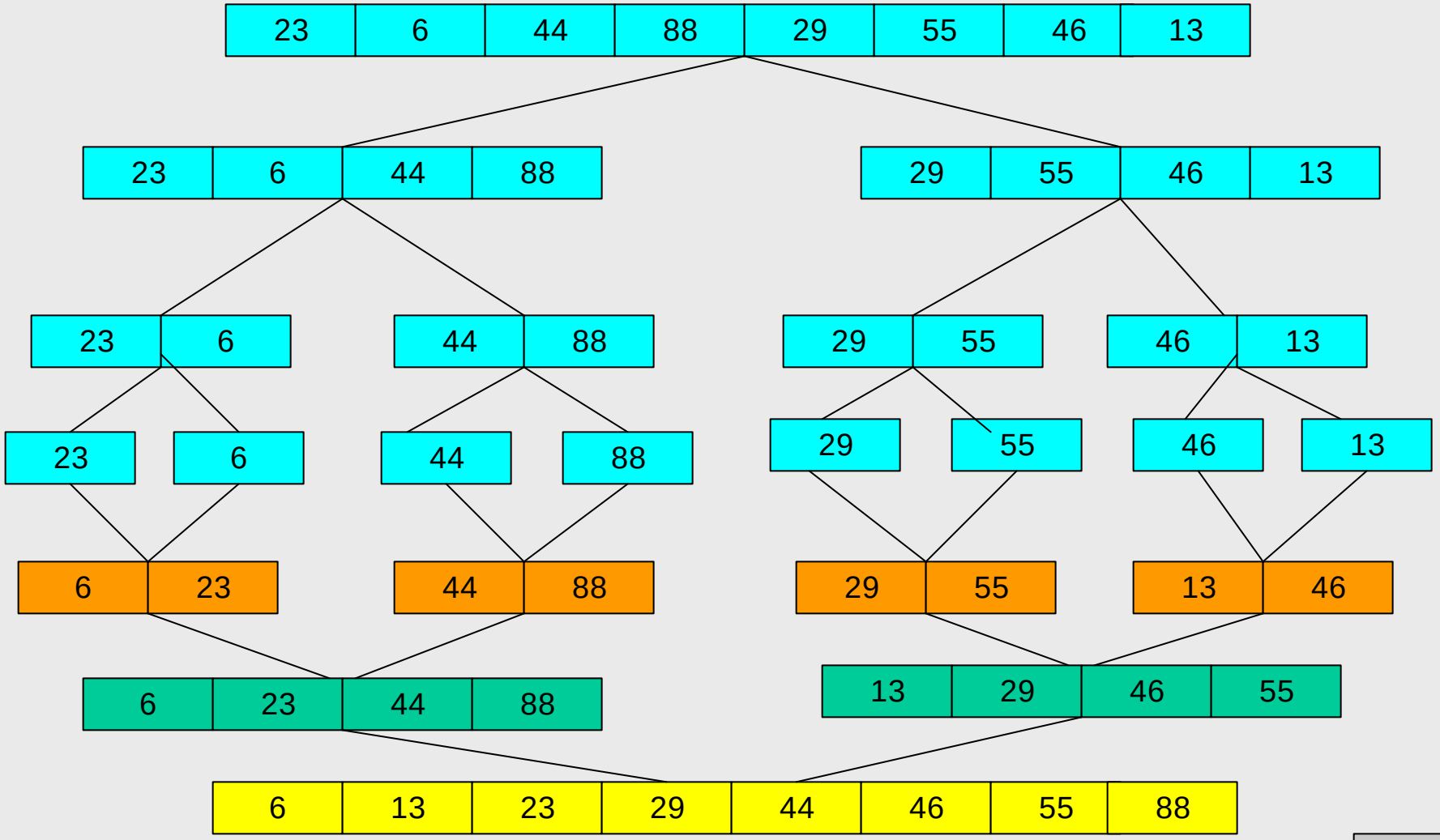

MergeLists Algorithm

```
MergeLists (list, start1, end1, start2, end2)
    // list       the elements to be put into order
    // start1   beginning of “list” A
    // end1     end of “list” A
    // start2   beginning of “list” B
    // end2     end of “list” B

    finalStart = start1;
    finalEnd = end1;
    indexC = 1;
    while (start1 <= end1) and (start2 <= end2) do
        if  list[start1] < list[start2] {
                result[indexC] = list[start1];
                start1++;}
         else {
                result[indexC] = list[start2];
                start2++;
        }
    indexC++;
```


MergeLists Algorithm Continues

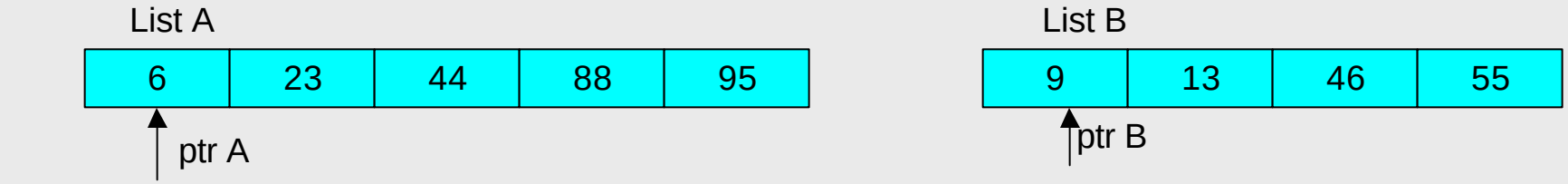
```
//move remainder of list that is left over
if (start1 <= end1) {
     for (i=start1; i <= end1; i++) {
            result[indexC] = list[i];
            indexC++;
     }
else
     for (i=start2; i <= end2, i++) {
             result[indexC] = list[i];
             indexC++;
     }

//put the result back into the original list
indexC = 1;
for (i = finalStart; i <= finalEnd; i++){
       list[i] = result[indexC];
       indexC++;
end.
```


Merge Sort Example

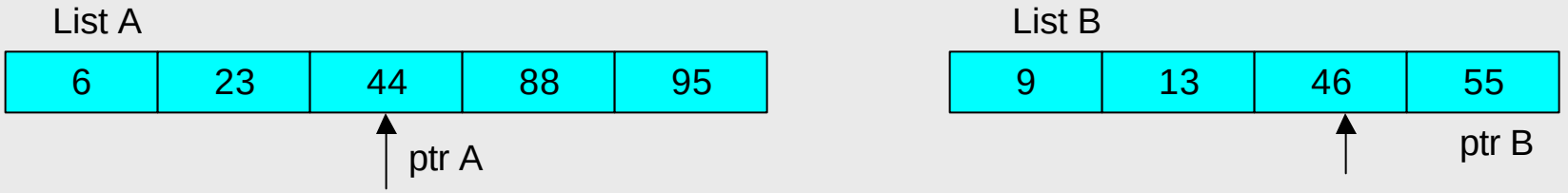


How MergeLists Works

initial configuration



configuration at 5 below:



1. Compare values of ptrA and ptrB: ptrA < ptrB, put 6 in final list, advance ptrA.
2. Compare values of ptrA and ptrB: ptrA > ptrB, put 9 in final list, advance ptrB.
3. Compare values of ptrA and ptrB: ptrA > ptrB, put 13 in final list, advance ptrB.
4. Compare values of ptrA and ptrB: ptrA < ptrB, put 23 in final list, advance ptrA.
5. Compare values of ptrA and ptrB: ptrA < ptrB, put 44 in final list, advance ptrA.
6. Compare values of ptrA and ptrB: ptrA > ptrB, put 46 in final list, advance ptrB.
7. Compare values of ptrA and ptrB: ptrA > ptrB, put 55 in final list, advance ptrB.
8. ptrB has reached the end of listB.
9. Add remainder of listA to final list.

MergeLists Analysis

- Because all of the element comparisons occur in MergeLists, we'll begin our analysis of MergeSort there.
- Let's look at the case when all of the elements of listA are smaller than the first element of listB. For example, listA= [2, 3, 4] and listB= [7, 8, 9, 10]. What will happen in MergeLists?
  - We'll compare A[1] with B[1] and since A[1] < B[1] it will move to the final result list.
  - Next, we'll compare A[2] with B[1] and since A[2] < B[1] it will move to the final result list.
- Since every element in listA will be compared with the first element of listB, this implies that the total number of comparisons will be equal to the number of elements in listA.

MergeLists Analysis (cont.)

- What will happen when the first element of listA is greater than the first element of listB but all of the remaining elements of listA are smaller than the second element of listB?
  - For example: listA = [4, 7, 8] and listB = [2, 10, 12, 14].
  - We'll compare A[1] and B[1] and move B[1] to the result list. Now we're in the same situation as before in that every remaining element in listA is smaller than B[2].
- In this case we'll need to compare every element of listA with B[2], plus the comparison we already did with A[1] and B[1]. In total we'll need a number of comparisons equal to the number of elements in listA plus 1.
- In general, MergeLists will require a total of $N_A + N_B - 1$ comparisons where N_A and N_B are the number of elements in listA and listB respectively.

MergeSort Analysis

- Now that we know the complexity of MergeLists, we can deal with MergeSort.
- Notice that the MergeSort is called recursively as long as first is less than last.
- If they are equal or first is greater than last, there is no recursive call.
  - If first is equal to last, then this represents a list of size 1. If first is greater than last, this represents a list of size 0. In both of these cases the algorithm does nothing, so the direct solution makes 0 comparisons.

MergeSort Analysis (cont.)

- The division of the list into two parts is done by the calculation of middle. This is done utilizing the halving principle which splits the list into two equal parts (integer division, so roughly equal).
- Thus a list with n elements will be split into a list of n/2 elements. From our earlier analysis of MergeLists, this means that the combine step requires n/2 comparisons in the best case, and n/2+n/2-1 or n-1 comparisons in the worst case.
- Putting this all of this together gives the two recurrence relations for the worst and best cases:
  - W(n) = 2W(n/2)+n-1
  - W(0) = W(1) = 0
  - B(n) = 2B(n/2)+n/2
  - B(0) = B(1) = 0

MergeSort Analysis (cont.)

- Solving for the worst case we have:
  - W(n) = 2W(n/2)+n-1
  - W(n/2) = 2W(n/4)+n/2-1
  - W(n/4) = 2W(n/8)+n/4-1
  - W(n/8) = 2W(n/16)+n/8-1
  - W(n/16) = 2W(n/32)+n/16-1
- Substituting:
  - W(n) = 2W(n/2)+n-1
  - W(n) = 2(2W(n/4)+n/2-1)+n-1
  - W(n) = 4W(n/4)+n-2+n-1
  - W(n) = 4(2W(n/8)+n/4-1)+n-2+n-1
  - W(n) = 8W(n/8)+n-4+n-2+n-1
  - W(n) = 8(2W(n/16)+n/8-1)+n-4+n-2+n-1
  - W(n) = 16W(n/16)+n-8+n-4+n-2+n-1
  - W(n) = 16(2W(n/32)+n/16-1)+n-8+n-4+n-2+n-1
  - W(n) = 32W(n/32)+n-16+n-8+n-4+n-2+n-1

MergeSort Analysis (cont.)

- The closed form of this equation becomes:

$$W(n) = n \times W(1) + n\log_2 n - \sum_{i=0}^{\log_2(n-1)} 2^i = n\log_2 n - n + 1 = O(n\log_2 n)$$

- In turns out that the best case performance of the merge sort algorithm is identical to this, which means that merge sort is a very efficient sort, even in the worst case.
- The main problem suffered by merge sort if the extra space required to support the merge.

Summary of Merge Sort

- Very efficient sorting algorithm.
- Suffers only from additional space required to support the merge. This is a very common external sorting algorithm.

Merge Sort	
best case	$O(n \log_2 n)$
average case	$O(n \log_2 n)$
worst case	$O(n \log_2 n)$