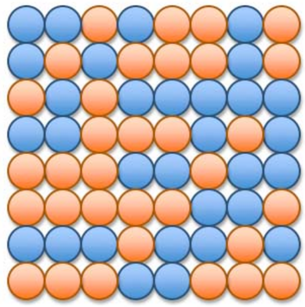


UNIT 5C
Merge Sort

Course Announcements

- Exam rooms for Lecture 1, 2:30 - 3:20
  - Sections A, B, C, D at Rashid
  - Sections E, F, G at Baker A51 (Giant Eagle Auditorium)
- Exam rooms for Lecture 2, 3:30 – 4:20
  - Sections H, I, J, K at Rashid
  - Sections L, M at PH125C
  - Section N at PH125B
- Bring your CMU id !

Divide and Conquer

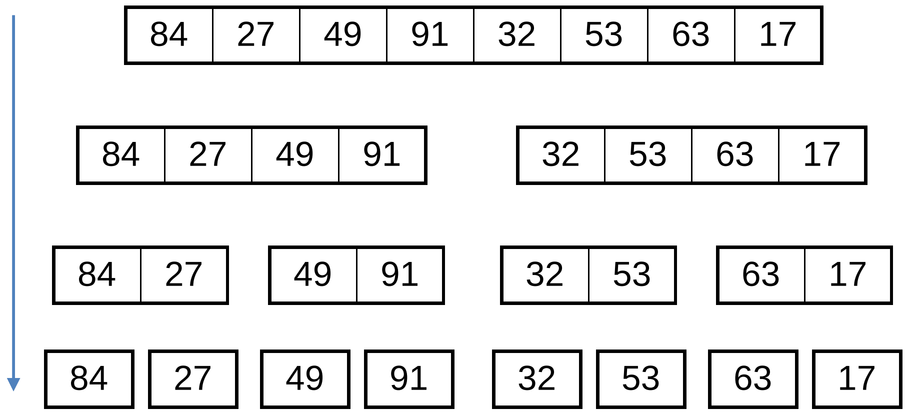
- In the military: strategy to gain or maintain power
- In computation:
  - Divide the problem into "simpler" versions of itself.
  - Conquer each problem using the same process (usually recursively).
  - Combine the results of the "simpler" versions to form your final solution.
- Examples: Towers of Hanoi, fractals, Binary Search, Merge Sort

Merge Sort

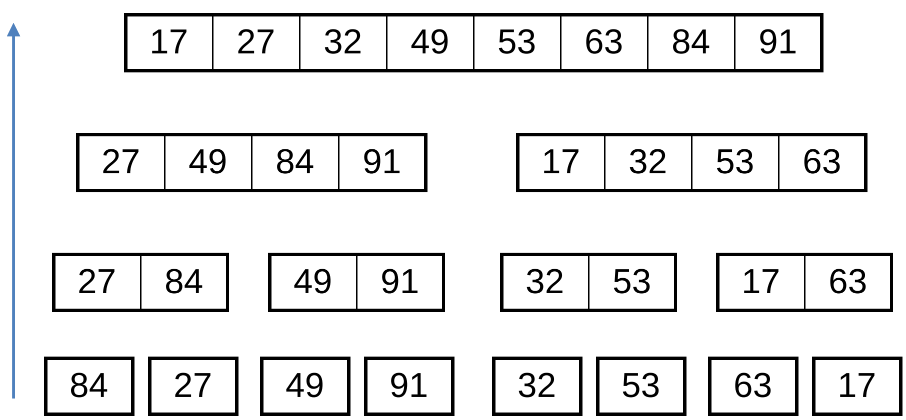
- Input: Array A of n elements.
- Result: Returns a new array containing the same elements in non-decreasing order.
- General algorithm for merge sort:

  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.

Divide (Split)



Conquer (Merge)



Example 1: Merge

```
array a        array b        array c
 0  1  2  3     0  1  2  3     0  1  2  3  4  5  6  7
12 44 58 62    29 31 74 80    12

 0  1  2  3     0  1  2  3     0  1  2  3  4  5  6  7
12 44 58 62    29 31 74 80    12 29

 0  1  2  3     0  1  2  3     0  1  2  3  4  5  6  7
12 44 58 62    29 31 74 80    12 29 31

 0  1  2  3     0  1  2  3     0  1  2  3  4  5  6  7
12 44 58 62    29 31 74 80    12 29 31 44
```

Example 1: Merge (cont'd)

```
array a        array b        array c
 0  1  2  3     0  1  2  3     0  1  2  3  4  5  6  7
12 44 58 62    29 31 74 80    12 29 31 44 58

 0  1  2  3     0  1  2  3     0  1  2  3  4  5  6  7
12 44 58 62    29 31 74 80    12 29 31 44 58 62

 0  1  2  3     0  1  2  3     0  1  2  3  4  5  6  7
12 44 58 62    29 31 74 80    12 29 31 44 58 62 74 80
```

Example 2: Merge

```
array a         array b         array c
 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
58 67 74 90     19 26 31 44     19

 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
58 67 74 90     19 26 31 44     19 26

 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
58 67 74 90     19 26 31 44     19 26 31

 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
58 67 74 90     19 26 31 44     19 26 31 44

 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
58 67 74 90     19 26 31 44     19 26 31 44 58 67 74 90
```

Merge

- Input: Two arrays a and b.
  - Each array must be sorted already in non-decreasing order.
- Result: Returns a new array containing the same elements merged together into a new array in non-decreasing order.
- We'll need two variables to keep track of where we are in arrays a and b: index_a and index_b.

1. Set index_a equal to 0.
2. Set index_b equal to 0.
3. Create an empty array c.

Merge (cont'd)

4. While index_a < the length of array a <u>and</u> index_b < the length of array b, do the following:

   a. If a[index_a] ≤ b[index_b], then do the following:

      i. append a[index_a] on to the end of array c

      ii. add 1 to index_a

   Otherwise, do the following:

      i. append b[index_b] on to the end of array c

      ii. add 1 to index_b

Merge (cont'd)

(Once we finish step 4, we've added all of the elements of either array a or array b to array c. The other array still has some elements left that need to be added to array c.)

5. If index_a < the length of array a, then:

   append all remaining elements of array a on to the end of array c

   Otherwise:

   append all remaining elements of array b on to the end of array c

6. Return array c as the result.

Merge in Ruby

```
def merge(a, b)
    index_a = 0
    index_b = 0
    c = []
    while index_a < a.length and index_b < b.length do
        if a[index_a] <= b[index_b] then
            c << a[index_a]
            index_a = index_a + 1
        else
            c << b[index_b]
            index_b = index_b + 1
        end
    end
```


Merge in Ruby (cont'd)

```
    if (index_a < a.length) then
        for i in (index_a..a.length-1) do
            c << a[i]
        end
    else
        for i in (index_b..b.length-1) do
            c << b[i]
        end
    end
    return c
end
```

Merge Sort: Base Case

- General algorithm for merge sort:
  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.
- What is the base case?
  If the list has only 1 element, it is already sorted so just return the list as the result.

Merge Sort: Halfway Point

- General algorithm for merge sort:
  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.
- How do we determine the halfway point where we want to split the array *list*?

  First half: 0..list.length/2-1

  Second half: list.length/2..list.length-1

Merge Sort in Ruby

```
def msort(list)
    return list if list.length == 1    # base case
    halfway = list.length/2
    list1 = list[0..halfway-1]
    list2 = list[halfway..list.length-1]
    newlist1 = msort(list1)         # recursive!
    newlist2 = msort(list2)         # recursive!
    newlist = merge(newlist1, newlist2)
    return newlist
end
```


Analyzing Efficiency

- If you merge two lists of size i/2 into one new list of size i, what is the maximum number of appends that you must do?
- Clearly, each element must be appended to the new list at some point, so the total number of appends is i.
- If you have a set of pairs of lists that need to be merged (two pairs at a time), and the total number of elements in all of the lists combined is n, the total number of appends will be n.

How many group merges?

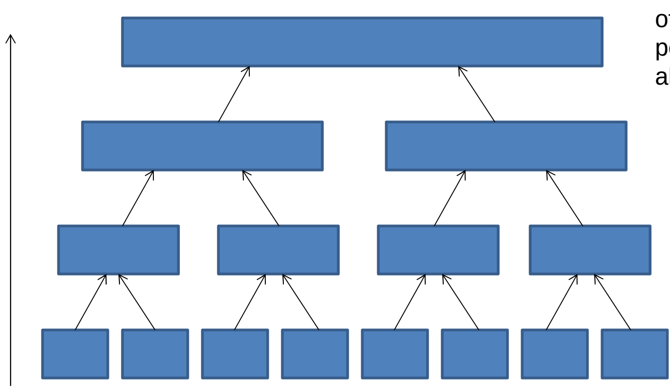
- How many group merges does it take to go from n groups of size 1 to 1 group of size n?
- Example: Merge sort on 32 elements.
  - Break down to groups of size 1 (base case).
  - Merge 32 lists of size 1 into 16 lists of size 2.
  - Merge 16 lists of size 2 into 8 lists of size 4.
  - Merge 8 lists of size 4 into 4 lists of size 8.
  - Merge 4 lists of size 8 into 2 lists of size 16.
  - Merge 2 lists of size 16 into 1 list of size 32.

  (the five merge steps above:) $5 = \log_2 32$
- In general: $\log_2 n$ group merges must occur.

Putting it all together

Total number of elements per level is always n.

It takes $\log_2 n$ iterations to go from n groups of size 1 to a single group of size n.



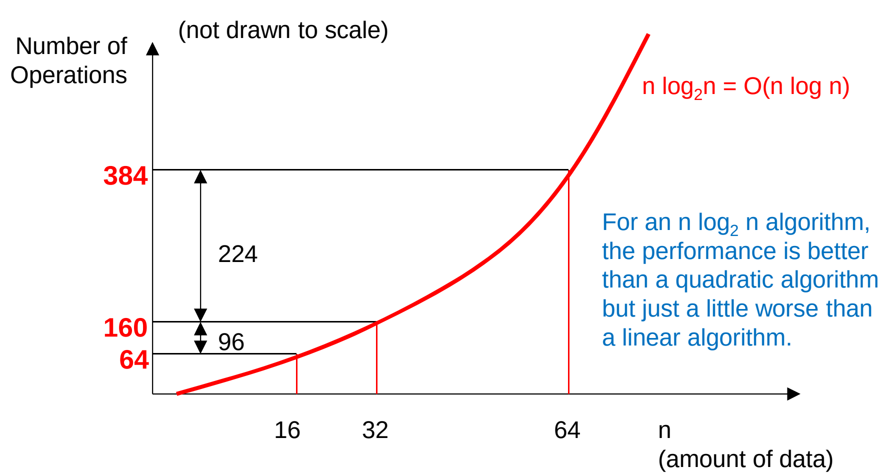
It takes n appends to merge all pairs to the next higher level.

Big O

- In the worst case, merge sort requires O(n log n) time to sort an array with n elements.

Number of operations	Order of Complexity
$n \log_2 n$	O(n log n)
$4n \log_{10} n$	O(n log n)
$n \log_2 n + 2n$	O(n log n)

O(N log N)



Comparing Insertion Sort to Merge Sort

(Worst Case)

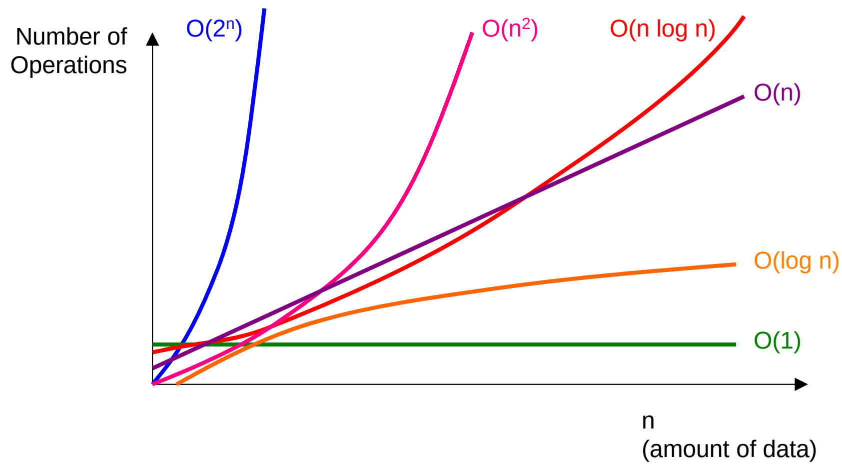
n	isort $(n(n+1)/2)$	msort $(n \log_2 n)$
8	36	24
16	136	64
32	528	160
2^{10}	524,800	10,240
2^{20}	549,756,338,176	20,971,520

For array sizes less than 100, there's not much difference between these sorts, but for larger arrays sizes, there is a clear advantage to merge sort.

Sorting and Searching

- Recall that if we wanted to use binary search, the array must be sorted.
  - What if we sort the array first using merge sort?
    - Merge sort O(n log n) (worst case)
    - Binary search O(log n) (worst case)
    - Total time: O(n log n) + O(log n) = O(n log n) (worst case)

Comparing Big O Functions



Merge Sort: Iteratively

(optional)

- *If you are interested, the textbook discusses an iterative version of merge sort which you can read on your own.*
- *This version uses an alternate version of the* `merge` *function that is not shown in the textbook but is given in the RubyLabs gem.*

Quick Sort

- Uses the technique of divide-and-conquer
  1. Pick a pivot
  2. Divide the array into two subarrays, those that are smaller and those that are greater
  3. Put the pivot in the middle, between the two sorted arrays