CS211FZ: Data Structures and Algorithms II

2- Divide and Conquer – Merge Sort

LECTURER: CHRIS ROADKNIGHT

CHRIS.ROADKNIGHT@MU.IE

Divide and Conquer

Real life examples?

Navigation – Suppose we want to drive from Fuzhou to Beijing

A. We could try and plan an entire route

B. We could divide the route into subroutes

- 1. Fuzhou to Jingdezhen
- 2. Jingdezhen to Anqing
- 3. Anqing to Hefei
- 4. Hefei to Xuzhou
- 5. Xuzhou to Jinan
- 6. Jinan to Beijing

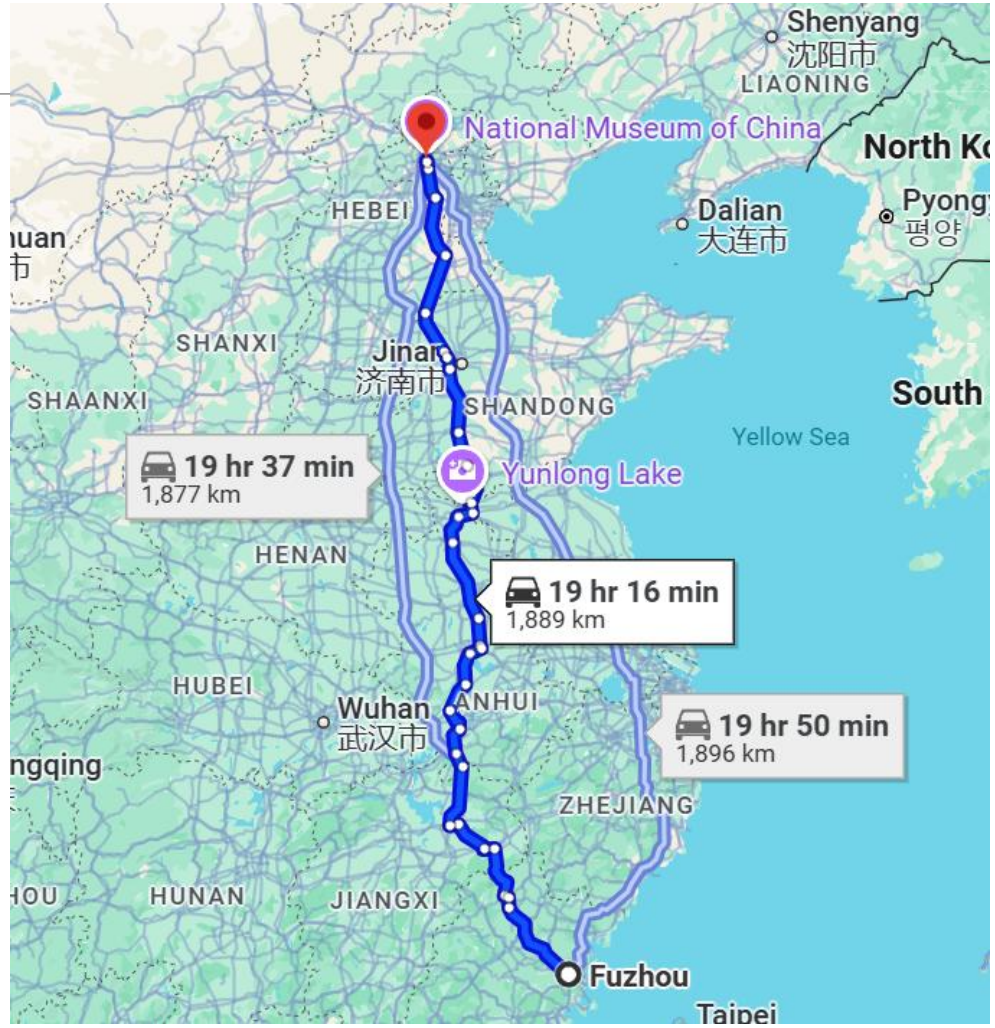
Why divide and conquer GPS mapping

If finding a route is exponentially complex

then breaking it down reduces the exponentiality.

It can also be allocated to different people to work out simultaneously

…but solution might be less good…why?



The divide-and-conquer approach via recursion

Many useful algorithms are **recursive** in structure: to solve a problem, they call themselves recursively one or more times to deal with closely related subproblems.

These algorithms break the problem into several subproblems that are similar to the original problem but **smaller** in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

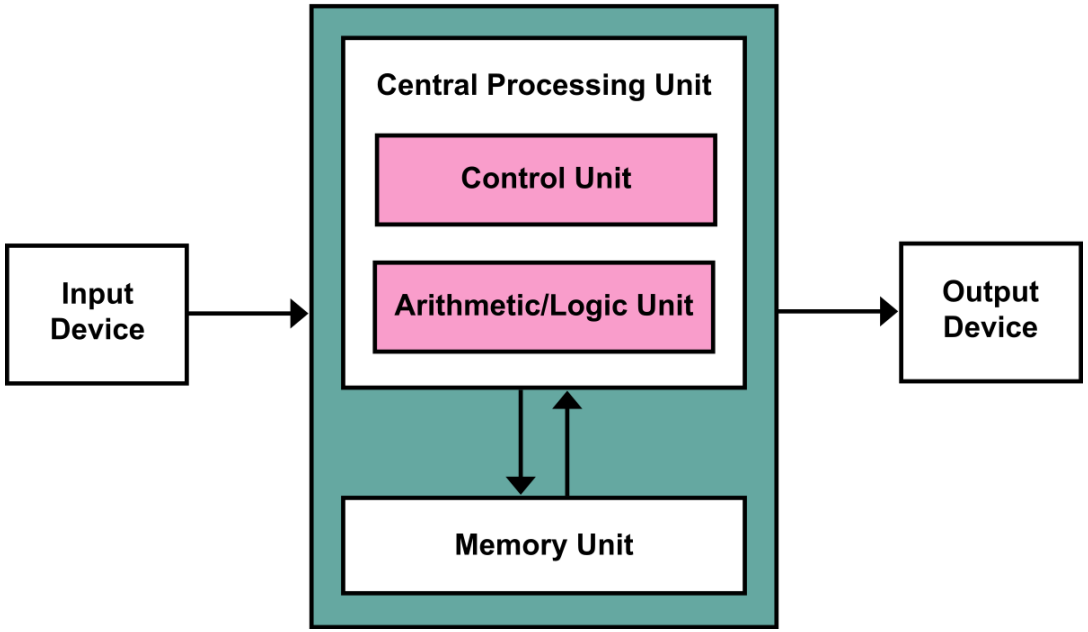
Three Steps in divide-and-conquer approach

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the sub problem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Merge Sort - https://en.wikipedia.org/wiki/Merge_sort



Merge Sort's famous inventer

MERGE SORT IS A DIVIDE-AND-CONQUER ALGORITHM THAT WAS INVENTED BY **JOHN VON NEUMANN** IN 1945.

The three Steps in merge sort

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two sub-sequences of $n/2$ elements each.

Conquer: Sort the two sub-sequences recursively using merge sort.

Combine: Merge the two sorted sub-sequences to produce the sorted answer.

Merge Sort Algorithm definition

Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$.

Output: A permutation (reordering) $< a'_1, a'_2, \ldots, a'_n >$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Idea: We break a list of numbers into a smaller size of sublists, solve each of them individually and merge the solutions to create a full solution to the problem.

Merge Sort Algorithm idea

The recursion reaches it's base case when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure $MERGE(A, p, q, r)$, where A is an array and p, q, and r are indices numbering elements of the array such that $p \leq q < r$. The procedure assumes that the subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray $A[p\,..\,r]$.

Merge Function - idea

Our MERGE procedure (next slides) takes time $\Theta(n)$, where $n = r - p + 1$ is the number of elements being merged, and it works as follows.

Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile.

Computationally, each basic step takes constant time, since we are checking just two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

Merge Function (1)

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
```

Merge Function (2)

```
11  // As long as each of the arrays L and R contains an unmerged element,
    //     copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //     remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Merge Function - Explanation (1)

In detail, the MERGE procedure works as follows.

It copies the two subarrays $A[p:q]$ and $A[q+1:r]$ into temporary arrays L and R ("left" and "right"), and then it merges the values in L and R back into $A[p:r]$.

Lines 1 and 2 compute the lengths n_L and n_R of the subarrays $A[p:q]$ and $A[q+1:r]$, respectively.

Then line 3 creates arrays $L[0:n_L-1]$ and $R[0:n_R-1]$ with respective lengths n_L and n_R. The ***for*** loop of lines 4-5 copies the subarray $A[p:q]$ into L, and the ***for*** loop of lines 6-7 copies the subarray $A[q+1:r]$ into R.

Merge Function - Explanation (2)

Lines 8-18, illustrated in Slides 13 and 14, perform the basic steps.

The ***while*** loop of lines 12-18 repeatedly identifies the smallest value in L and R that has yet to be copied back into $A[p:r]$ and copies it back in.

As the comments indicate, the index k gives the position of A that is being filled in, and the indices i and j give the positions in L and R, respectively, of the smallest remaining values.

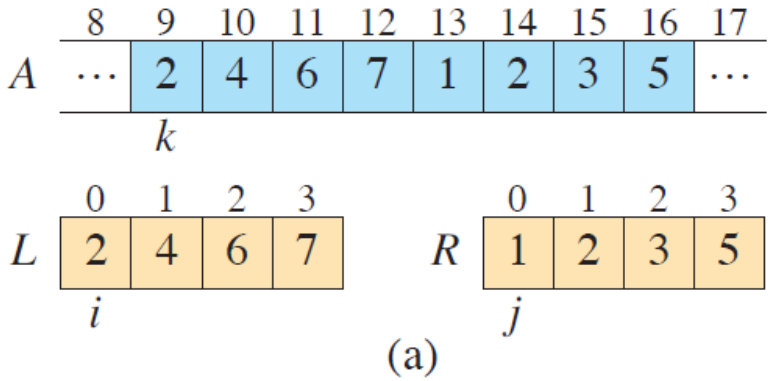
Eventually, either all of L or all of R is copied back into $A[p:r]$, and this loop terminates.

Merge Function - Explanation (3)

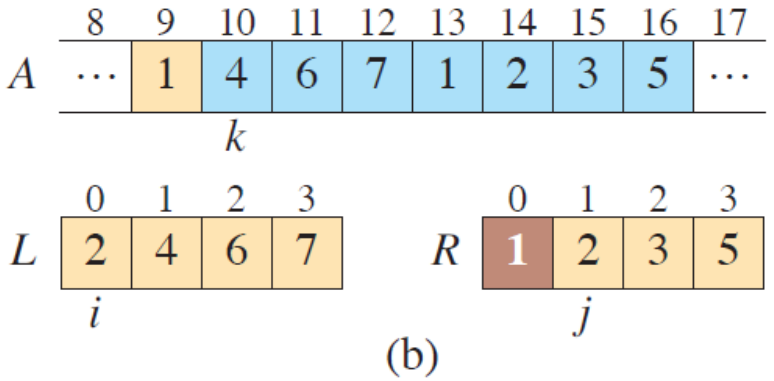
If the loop terminates because all of R has been copied back, that is, because j equals n_R , then i is still less than n_L, so that some of L has yet to be copied back, and these values are the greatest in both L and R. In this case, the ***while*** loop of lines 20-23 copies these remaining values of L into the last few positions of $A[p:r]$.

Because j equals n_R, the ***while*** loop of lines 24-27 iterates 0 times. If instead the ***while*** loop of lines 12-18 terminates because i equals n_L, then all of L has already been copied back into $A[p:r]$, and the ***while*** loop of lines 24-27 copies the remaining values of R back into the end of $A[p:r]$.

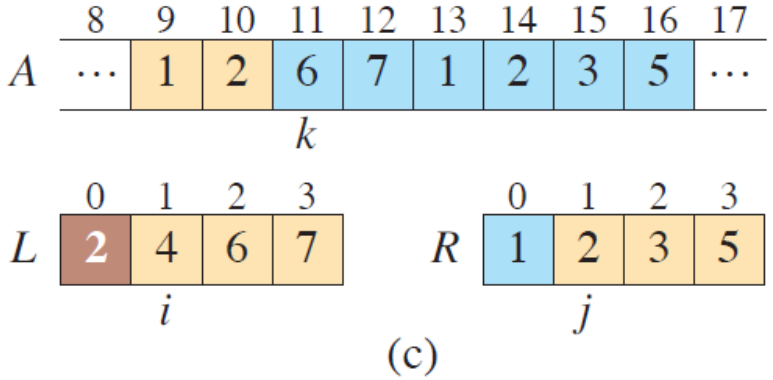
Merge Function - Example (1)



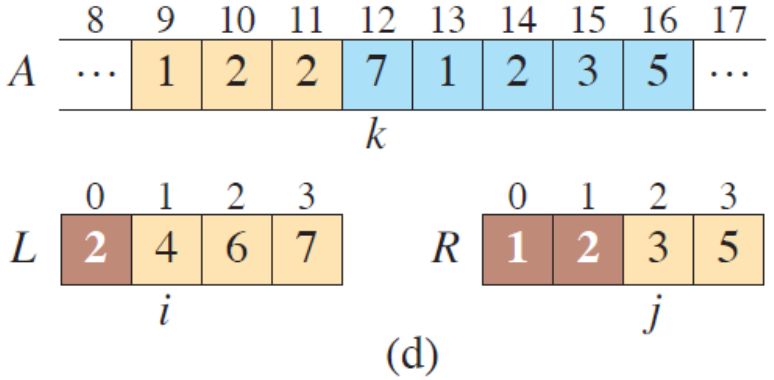
(a)



(b)

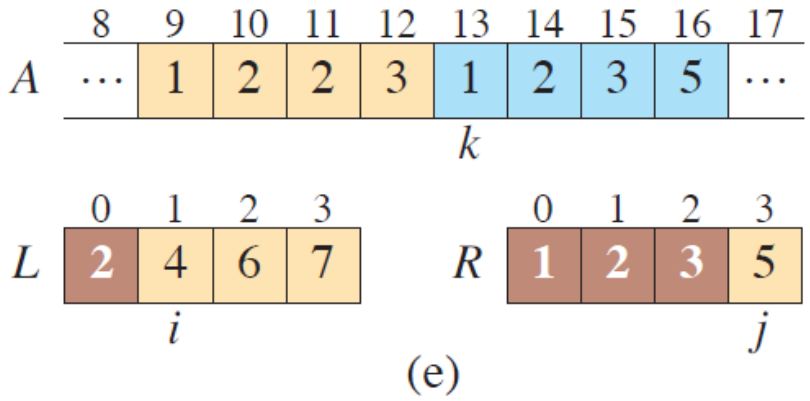


(c)

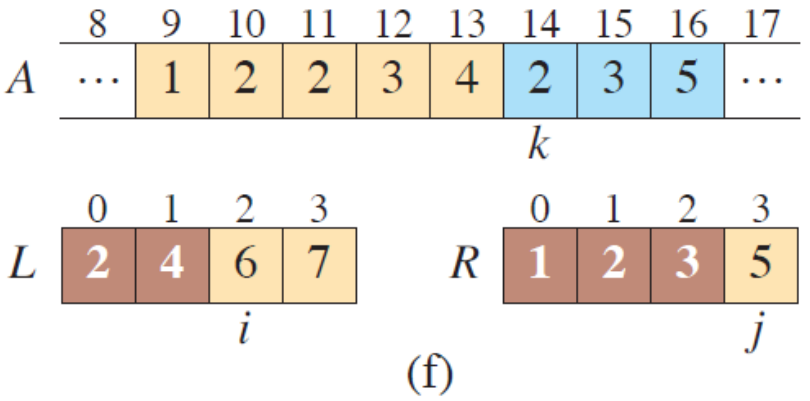


(d)

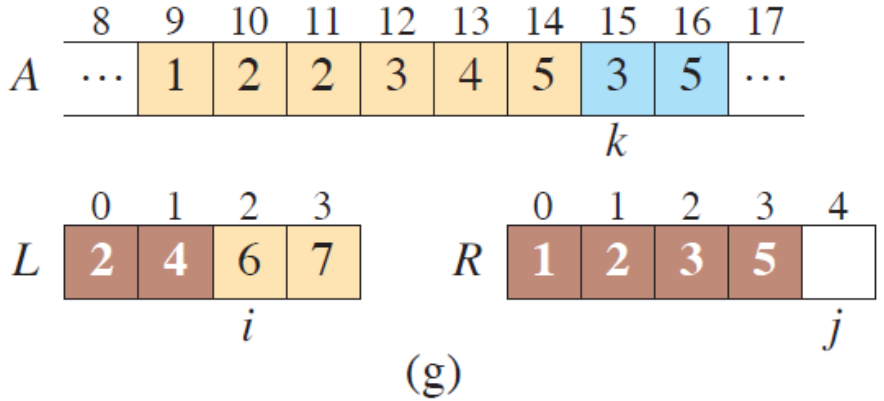
Merge Function - Example (2)



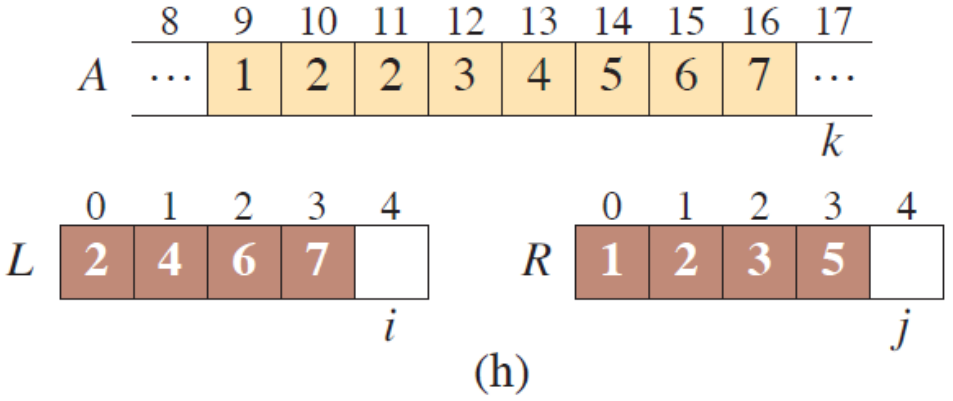
(e)



(f)



(g)



(h)

Merge Function - Example Explanation (1)

The operation of the ***while*** loop in lines 8-18 in the call $MERGE(A, 9, 12, 16)$, when the subarray $A[9: 16]$ contains the values $(2, 4, 6, 7, 1, 2, 3, 5)$.

After allocating and copying into the arrays L and R, the array L contains (2, 4, 6, 7), and the array R contains (1, 2, 3, 5).

Tan positions in A contain their final values, and tan positions in L and R contain values that have yet to be copied back into A. Taken together, the tan positions always comprise the values originally in $A[9: 16]$.

Blue positions in A contain values that will be copied over, and dark positions in L and R contain values that have already been copied back into A.

Merge Function - Example Explanation (2)

(a)–(g) The arrays A, L, and R, and their respective indices k, i, and j prior to each iteration of the loop of lines 12-18.

At the point in part (**g**), all values in R have been copied back into A (indicated by j equalling the length of R), and so the ***while*** loop in lines 12-18 terminates.

(h) The arrays and indices at termination.

The ***while*** loops of lines 20-23 and 24-27 copied back into A the remaining values in L and R, which are the largest values originally in $A[9:16]$. Here, lines 20-23 copied $L[2:3]$ into $A[15:16]$, and because all values in R had already been copied back into A, the ***while*** loop of lines 24-27 iterated 0 times. At this point, the subarray in $A[9:16]$ is sorted.

Merge Function – run time

To see that the MERGE procedure runs in $\theta(n)$ time, where $n = r - p + 1$, observe that each of lines 1-3 and 8-10 takes constant time, and the ***for*** loops of lines 4-7 take $\theta(n_L + n_R) = \theta(n)$ time.

To account for the three ***while*** loops of lines 12-18, 20-23, and 24-27, observe that each iteration of these loops copies exactly one value from L or R back into A and that every value is copied back into A exactly once.

Therefore, these three loops together make a total of n iterations. Since each iteration of each of the three loops takes constant time, the total time spent in these three loops is $\theta(n)$.

Merge-Sort Function (1)

We can now use the MERGE procedure as a subroutine in the merge sort algorithm.

The procedure $MERGE - SORT(A, p, r)$ on the facing page sorts the elements in the subarray $A[p:r]$. If p equals r, the subarray has just 1 element and is therefore already sorted.

Otherwise, we must have $p < r$, and MERGE-SORT runs the divide, conquer, and combine steps.

The divide step simply computes an index q that partitions $A[p:r]$ into two adjacent subarrays: $A[p:q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1:r]$, containing $\lfloor n/2 \rfloor$ elements.

Merge-Sort Function (2)

```
MERGE-SORT(A, p, r)
1  if p ≥ r                                  // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                           // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)                       // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)                   // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

The initial call $MERGE - SORT(A, 1, n)$ sorts the entire array $A[1:n]$.

Merge-Sort - Example

Next slide illustrates the operation of the procedure for $n = 8$, showing also the sequence of divide and merge steps.

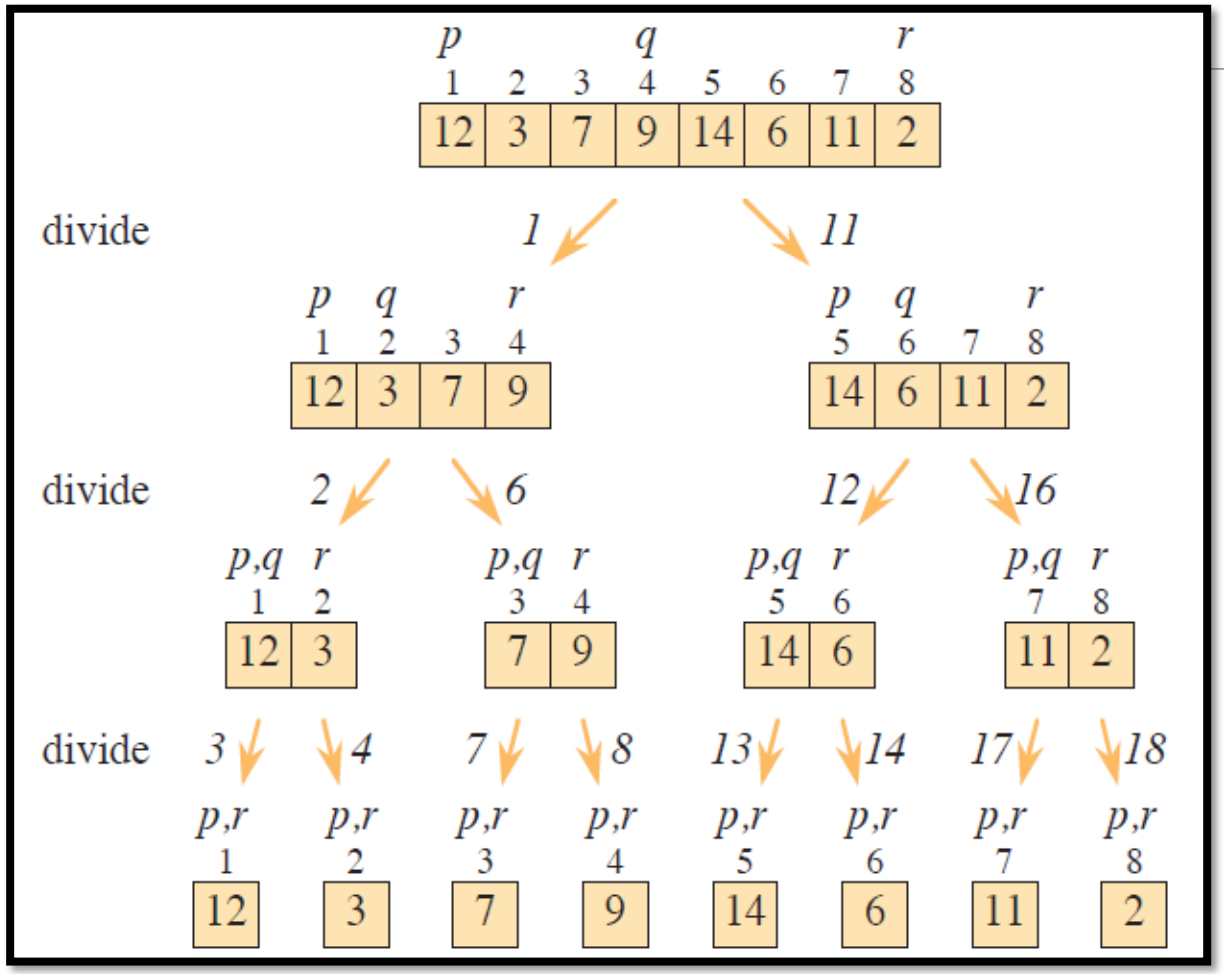
The algorithm recursively divides the array down to 1-element subarrays. The combine steps merge pairs of 1-element subarrays to form sorted subarrays of length 2, merges those to form sorted subarrays of length 4, and merges those to form the final sorted subarray of length 8.

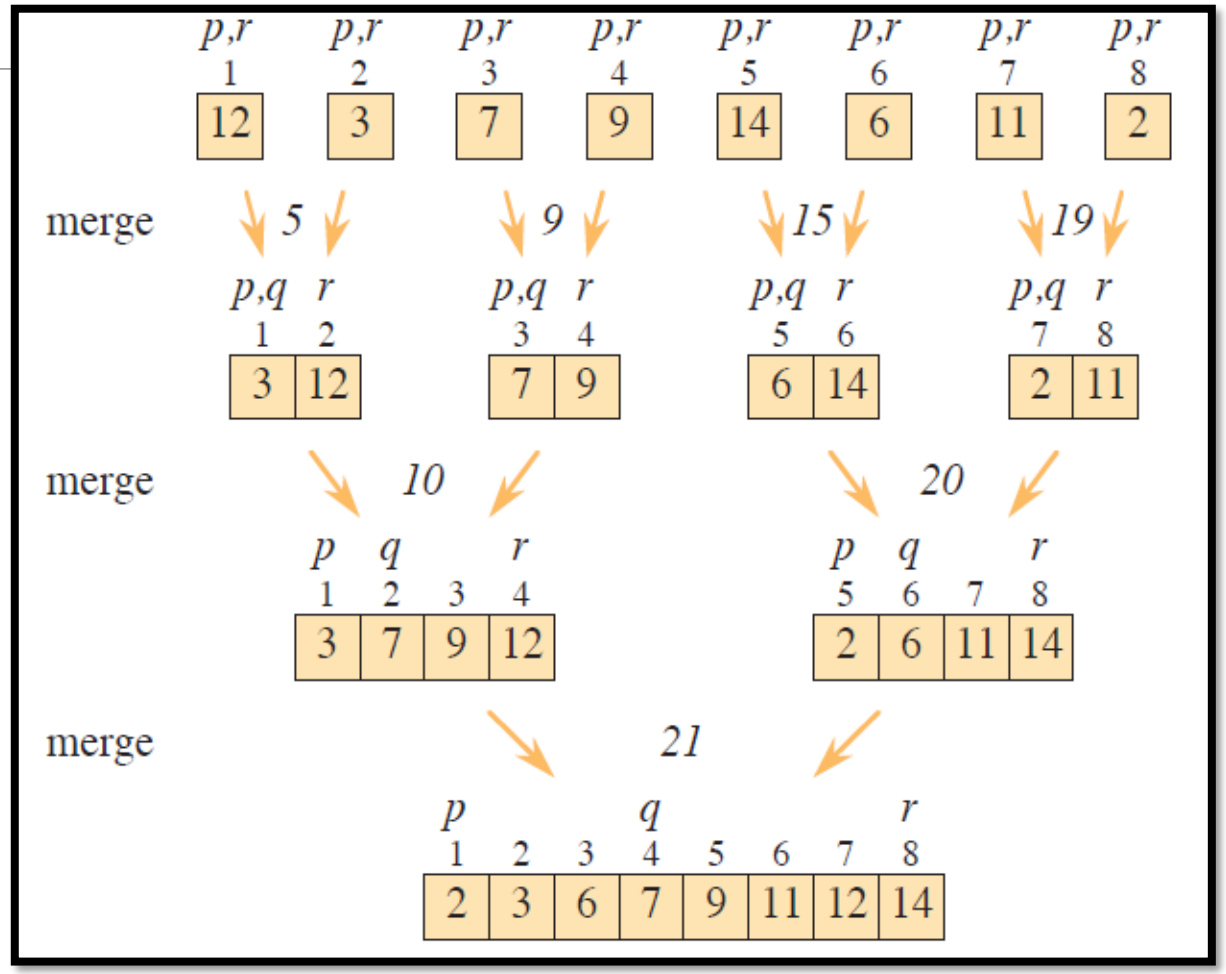
If n is not an exact power of 2, then some divide steps create subarrays whose lengths differ by 1. (For example, when dividing a subarray of length 7, one subarray has length 4 and the other has length 3.)

Regardless of the lengths of the two subarrays being merged, the time to merge a total of n items is $\theta(n)$.

1- Dividing

2- Merging





Insertion vs merge sort

Which approach would sort into increasing frequency fastest if we started with an array of the sequence 9,8,7,6,5,4,3,2,1???

Which approach would sort into increasing frequency fastest if we started with an array of the sequence 1,2,3,4,5,6,7,8,9???