Divide-and-conquer

Merge Sort

Dr. Bibhudatta Sahoo
Communication & Computing Group
Department of CSE, NIT Rourkela
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Merge sort

Definition:

Merge sort is a sort algorithm that splits the items to be sorted into two groups, recursively sorts each group, and merges them into a final sorted sequence.

Features:

- Is a comparison based algorithm
- Is a stable algorithm
- Is a perfect example of divide & conquer algorithm design strategy
- It was invented by John Von Neumann

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $\boldsymbol{S}$ in two disjoint subsets $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Recur: solve the subproblems associated with $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Conquer: combine the solutions for $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a solution for $\boldsymbol{S}$
- The base case for the recursion are subproblems of size 0 or 1

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - It has $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

```
Algorithm mergeSort(S, C)
    Input sequence S with n
        elements, comparator C
    Output sequence S sorted
        according to C
    if S.size() > 1
        (S_1, S_2) ← partition(S, n/2)
        mergeSort(S_1, C)
        mergeSort(S_2, C)
        S ← merge(S_1, S_2)
```

Merge-Sort

- MergeSort is a recursive sorting procedure that uses at most **O(n lg(n))** comparisons.
- To sort an array of **n** elements, we perform the following steps in sequence:
- If **n < 2** then the array is already sorted.
- Otherwise, **n > 1**, and we perform the following three steps in sequence:

1. **Sort** the **left** **half** of the the array using MergeSort.
2. **Sort** the **right** **half** of the the array using MergeSort.
3. **Merge** the sorted left and right halves.

Merge-Sort

- The performance of any divide-and-conquer algorithm will be good if the resultant subproblems are as evenly sized as possible.

- Algorithm MERGESORT sorts an array of n elements in time $\theta(n \log n)$ and space $\theta(n)$.

Performance of Merge Sort

- The performance of any divide-and-conquer algorithm will be good if the resultant sub-problems are as evenly sized as possible.
- Let T(N) denote the worst-case running time of mergesort to sort N numbers.
- Assume that N is a power of 2.
- Divide step: O(1) time
- Conquer step: 2 T(N/2) time
- Combine step: O(N) time
- Recurrence equation:

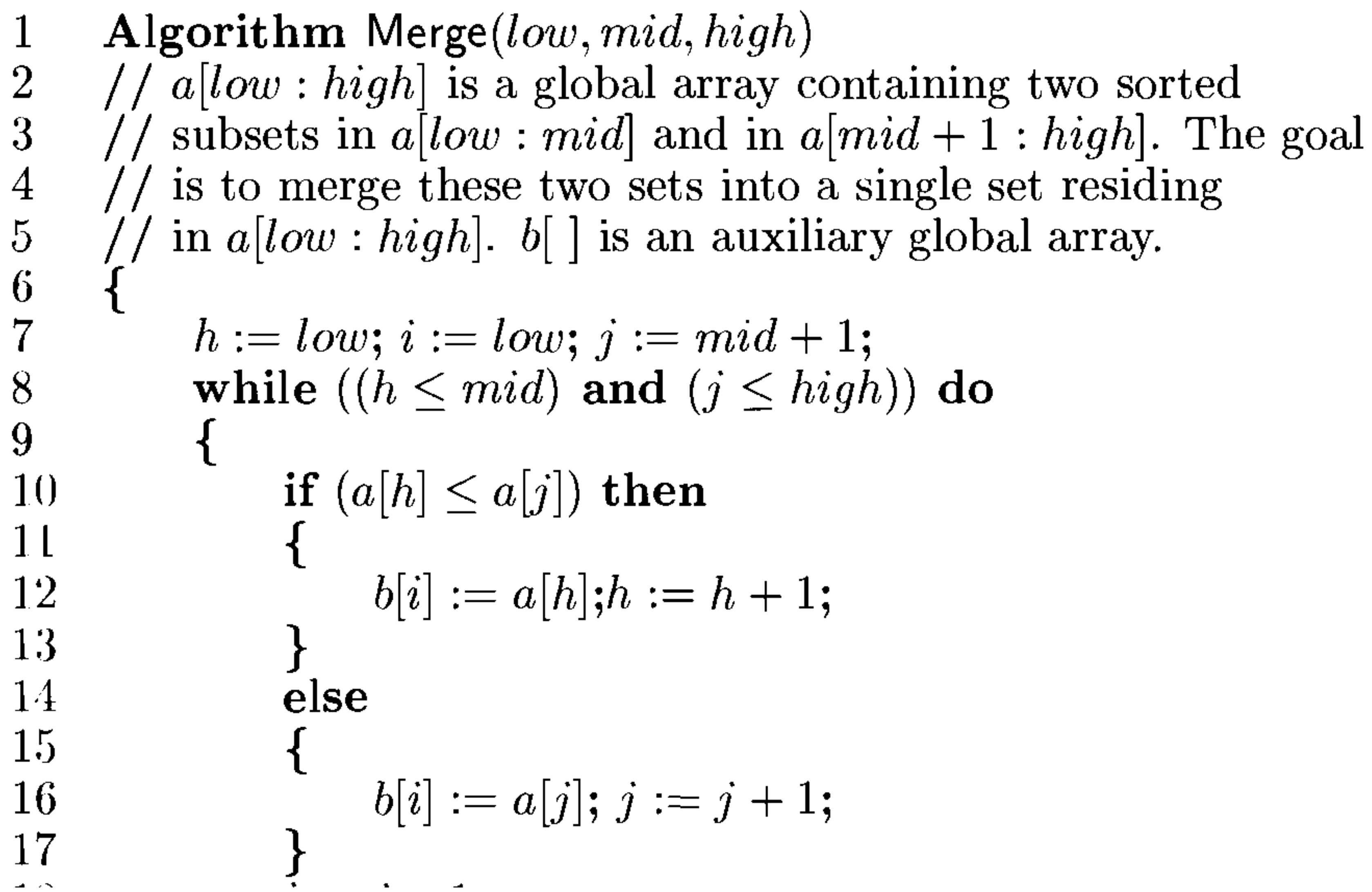
$$T(1) = 1 \quad ; \text{if } N < 2$$
$$T(N) = 2T(N/2) + N \quad ; \text{if } N \geq 2$$

Q. Algorithm MERGESORT sorts an array of n elements in time θ(n log n) and space θ (n).

Merge sort

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then // If there are more than one element
    {
        // Divide P into subproblems.
        // Find where to split the set.
        mid:= $\lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
        MergeSort(low, mid);
        MergeSort(mid + 1, high);
        // Combine the solutions.
        Merge(low, mid, high);
    }
}
```

Algorithm Merge(low, mid, high) : 1/2



```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ is a global array containing two sorted
// subsets in $a[low : mid]$ and in $a[mid + 1 : high]$. The goal
// is to merge these two sets into a single set residing
// in $a[low : high]$. $b[\,]$ is an auxiliary global array.
{
    $h := low$; $i := low$; $j := mid + 1$;
    while (($h \leq mid$) and ($j \leq high$)) do
    {
        if ($a[h] \leq a[j]$) then
        {
            $b[i] := a[h]$;$h := h + 1$;
        }
        else
        {
            $b[i] := a[j]$; $j := j + 1$;
        }
```

Algorithm Merge(low, mid, high): 2/2

```
            $i := i + 1;$
        }
        if $(h > mid)$ then
            for $k := j$ to $high$ do
            {
                $b[i] := a[k]; i := i + 1;$
            }
        else
            for $k := h$ to $mid$ do
            {
                $b[i] := a[k]; i := i + 1;$
            }
        for $k := low$ to $high$ do $a[k] := b[k];$
}
```

Procedure Merge

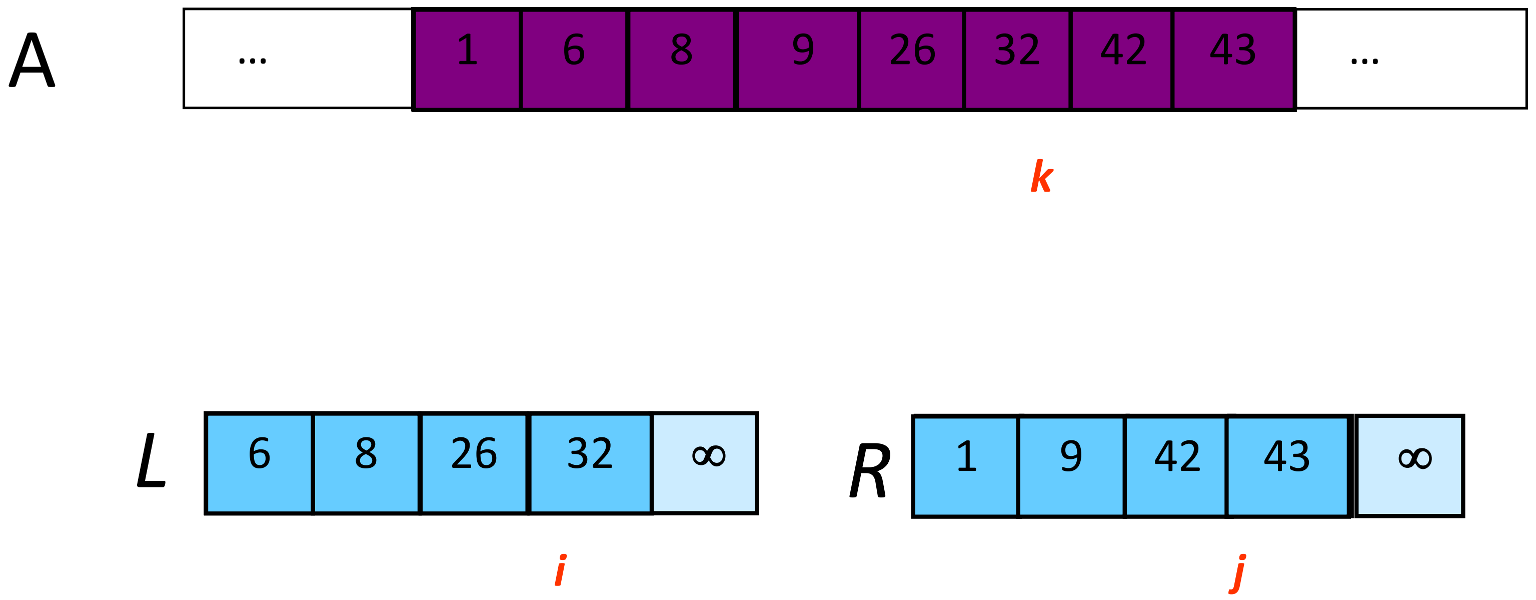
```
Merge(A, p, q, r)
1   $n_1 \leftarrow q - p + 1$
2   $n_2 \leftarrow r - q$
3       for $i \leftarrow 1$ to $n_1$
4           do $L[i] \leftarrow A[p + i - 1]$
5       for $j \leftarrow 1$ to $n_2$
6           do $R[j] \leftarrow A[q + j]$
7       $L[n_1+1] \leftarrow \infty$
8       $R[n_2+1] \leftarrow \infty$
9       $i \leftarrow 1$
10      $j \leftarrow 1$
11      for $k \leftarrow p$ to $r$
12          do if $L[i] \leq R[j]$
13              then $A[k] \leftarrow L[i]$
14                  $i \leftarrow i + 1$
15              else $A[k] \leftarrow R[j]$
16                  $j \leftarrow j + 1$
```

Input: Array containing sorted subarrays *A*[*p*..*q*] and *A*[*q*+1..*r*].

Output: Merged sorted subarray in *A*[*p*..*r*].

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge – Example



Correctness of Merge

```
Merge(A, p, q, r)
1  $n_1 \leftarrow q - p + 1$
2  $n_2 \leftarrow r - q$
3      for $i \leftarrow 1$ to $n_1$
4          do $L[i] \leftarrow A[p + i - 1]$
5      for $j \leftarrow 1$ to $n_2$
6          do $R[j] \leftarrow A[q + j]$
7      $L[n_1+1] \leftarrow \infty$
8      $R[n_2+1] \leftarrow \infty$
9      $i \leftarrow 1$
10     $j \leftarrow 1$
11     for $k \leftarrow p$ to $r$
12         do if $L[i] \leq R[j]$
13             then $A[k] \leftarrow L[i]$
14                 $i \leftarrow i + 1$
15             else $A[k] \leftarrow R[j]$
16                 $j \leftarrow j + 1$
```

Loop Invariant for the *for* loop

At the start of each iteration of the for loop:

Subarray $A[p..k - 1]$ contains the $k - p$ smallest elements of L and R in sorted order.

$L[i]$ and $R[j]$ are the smallest elements of L and R that have not been copied back into A.

Initialization:

Before the first iteration:

- $A[p..k - 1]$ is empty.
- $i = j = 1$.
- $L[1]$ and $R[1]$ are the smallest elements of L and R not copied to A.

Correctness of Merge

```
Merge(A, p, q, r)
1   $n_1 \leftarrow q - p + 1$
2   $n_2 \leftarrow r - q$
3       for $i \leftarrow 1$ to $n_1$
4           do $L[i] \leftarrow A[p + i - 1]$
5       for $j \leftarrow 1$ to $n_2$
6           do $R[j] \leftarrow A[q + j]$
7       $L[n_1+1] \leftarrow \infty$
8       $R[n_2+1] \leftarrow \infty$
9       $i \leftarrow 1$
10      $j \leftarrow 1$
11      for $k \leftarrow p$ to $r$
12          do if $L[i] \leq R[j]$
13              then $A[k] \leftarrow L[i]$
14                      $i \leftarrow i + 1$
15              else $A[k] \leftarrow R[j]$
16                      $j \leftarrow j + 1$
```

Maintenance:

Case 1: $L[i] \leq R[j]$

- By LI, A contains $p - k$ smallest elements of L and R in sorted order.
- By LI, $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied into A.
- Line 13 results in A containing $p - k + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

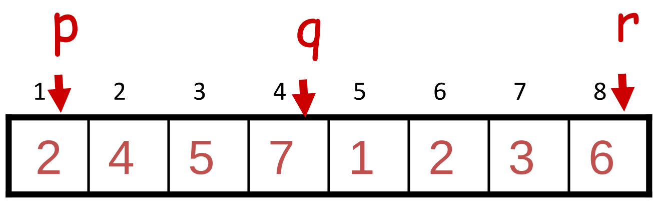
Similarly for $L[i] > R[j]$.

Termination:

- On termination, $k = r + 1$.
- By LI, A contains $r - p + 1$ smallest elements of L and R in sorted order.
- L and R together contain $r - p + 3$ elements. All but the two sentinels have been copied back into A.

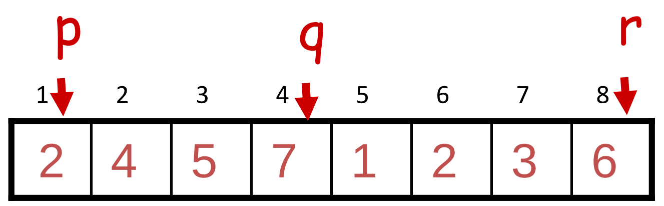
Working of Merg algorithm

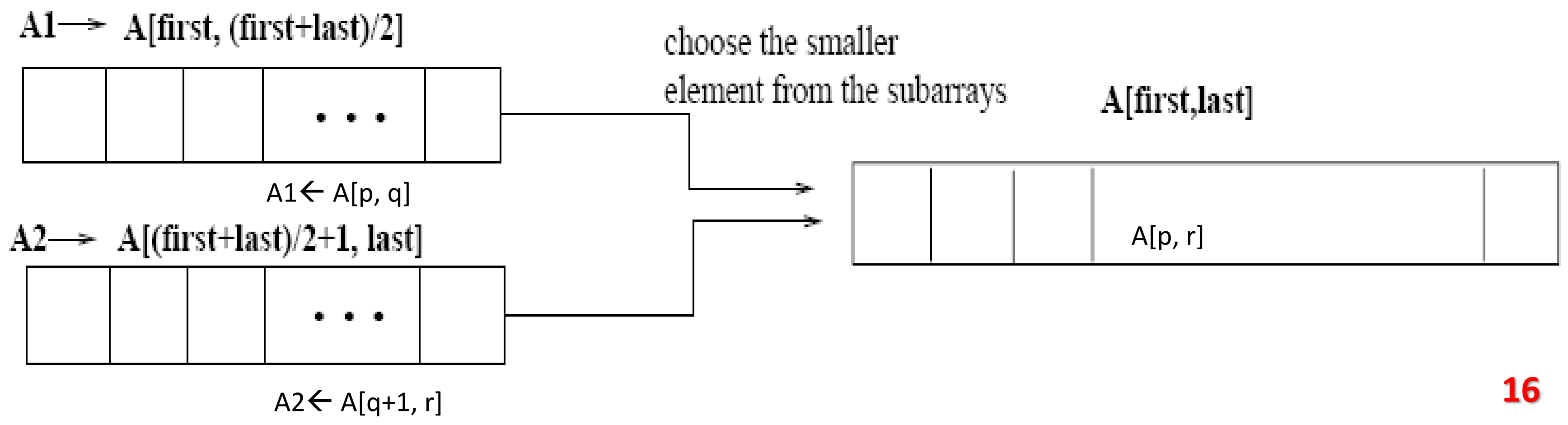
- **Input:** Array *A* and indices **p, q, r** such that **p ≤ q < r**
  - Subarrays *A*[p . . q] and *A*[q + 1 . . r] are sorted
- **Output:** One single sorted subarray *A*[p . . r]



Merging

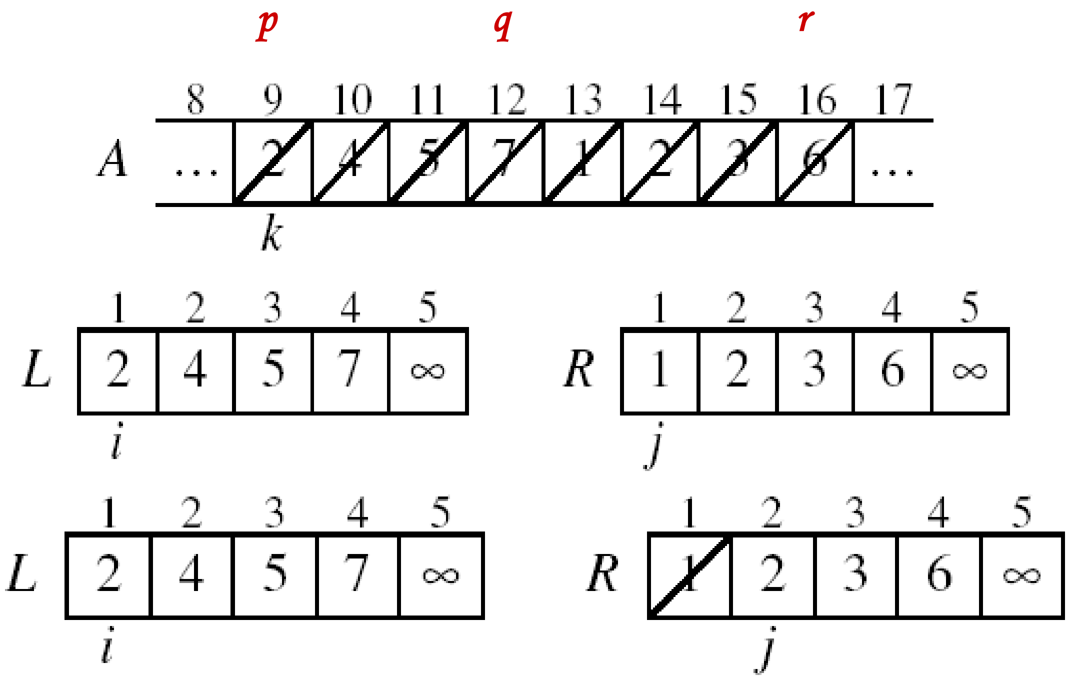
- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile



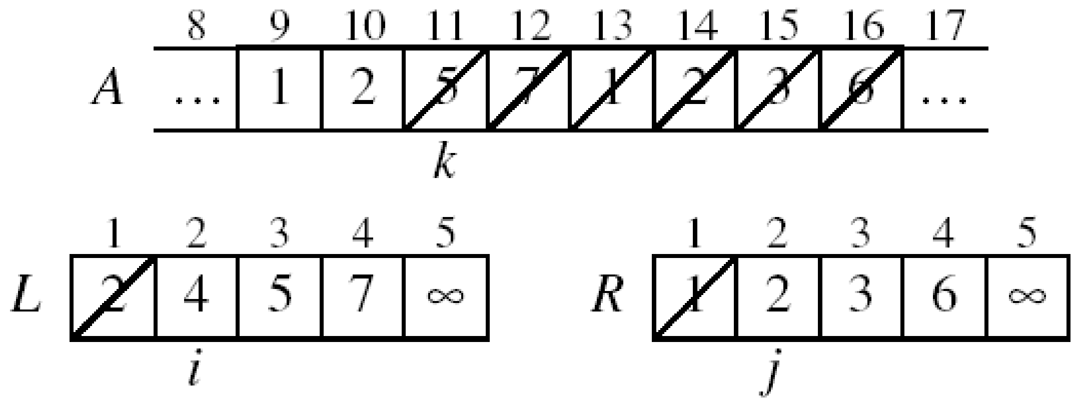


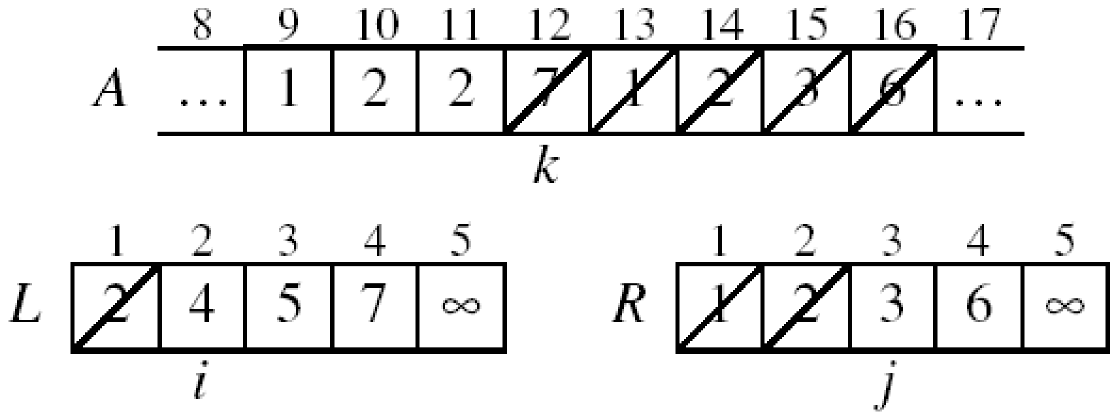
A1 → A[first, (first+last)/2]
A1← A[p, q]
A2 → A[(first+last)/2+1, last]
A2← A[q+1, r]
choose the smaller element from the subarrays
A[first,last]
A[p, r]

Example: MERGE(A, 9, 12, 16)

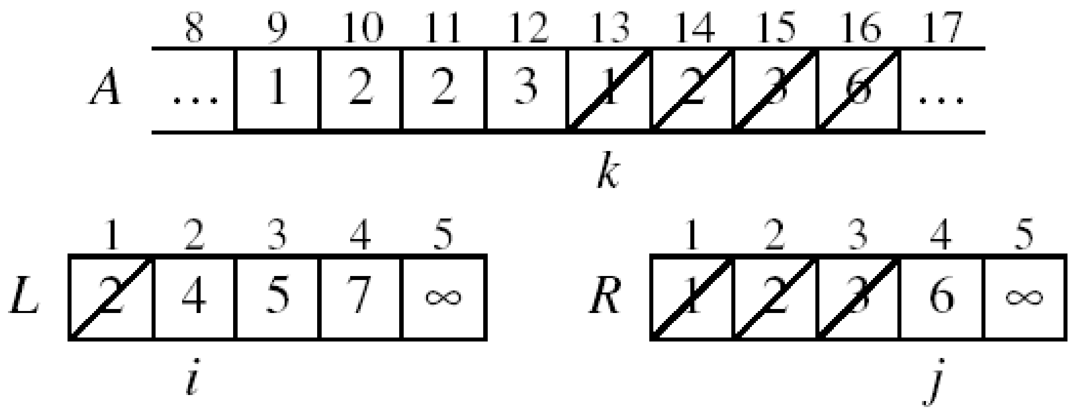


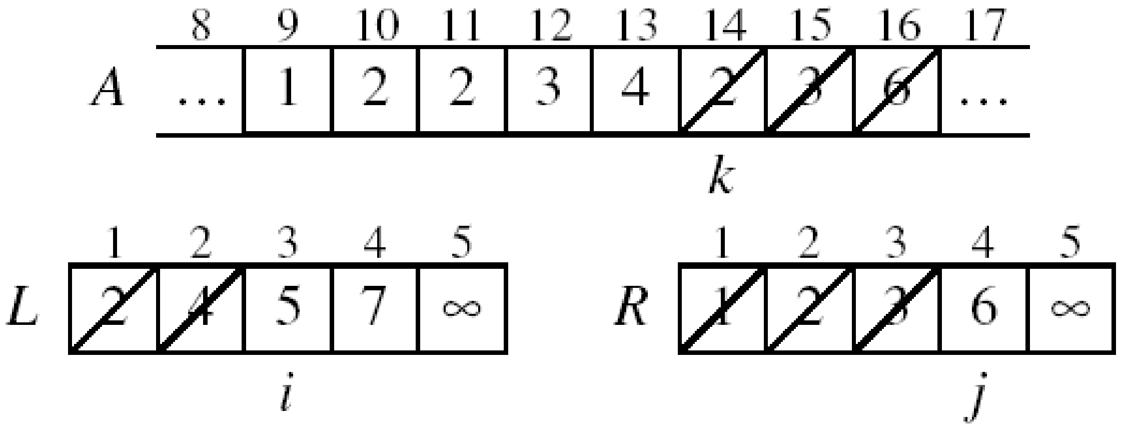
Example: MERGE(A, 9, 12, 16)



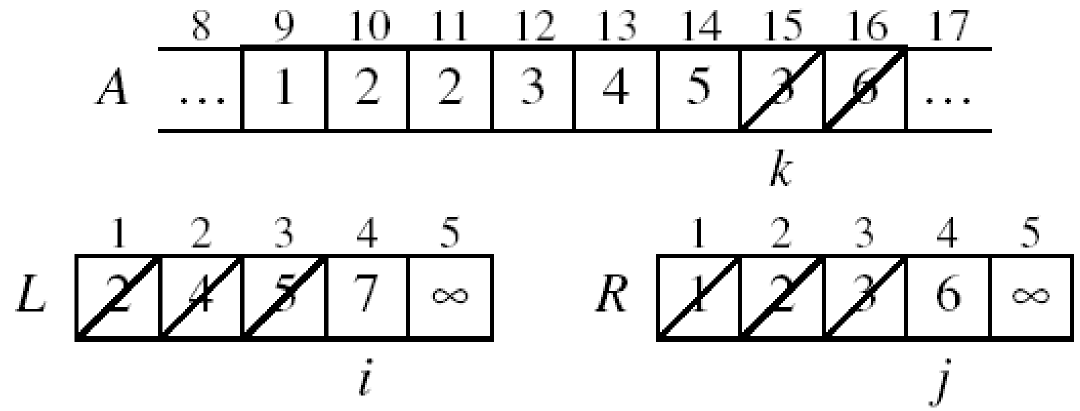


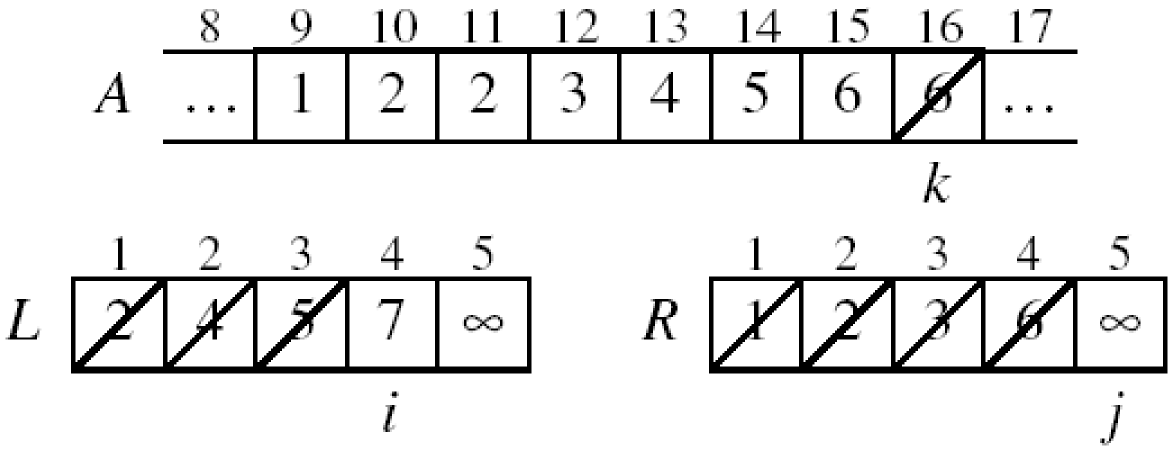
Example (cont.)



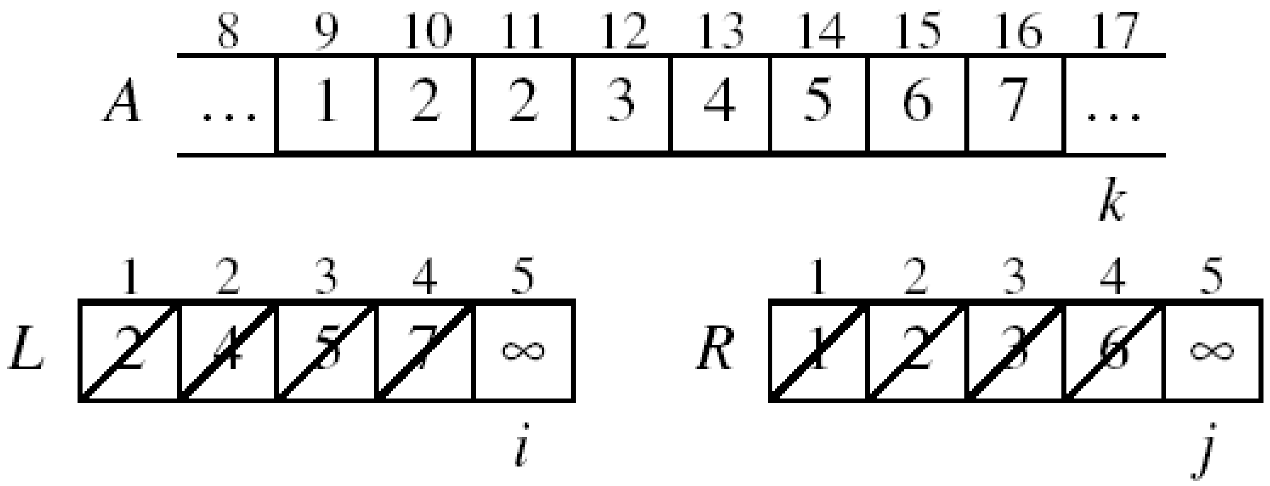


Example (cont.)





Example (cont.)

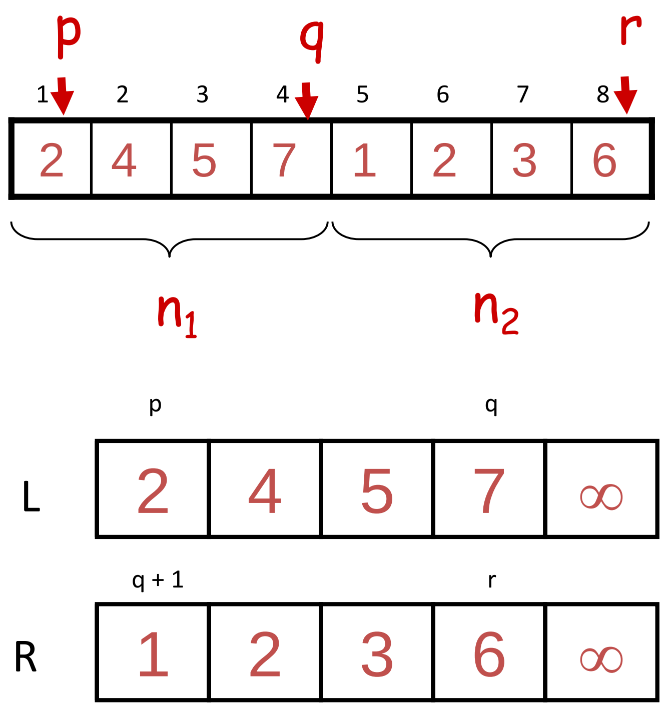


Done!

Merge - Pseudocode

Algorithm: MERGE(A, p, q, r)

```
1.  Compute $n_1$ and $n_2$
2.  Copy the first $n_1$ elements into
    $L[1 .. n_1 + 1]$ and the next $n_2$ elements into $R[1 .. n_2 + 1]$
3.  $L[n_1 + 1] \leftarrow \infty$;    $R[n_2 + 1] \leftarrow \infty$
4.  $i \leftarrow 1$;    $j \leftarrow 1$
5.  for $k \leftarrow p$ to $r$
6.      do if $L[ i ] \le R[ j ]$
7.          then $A[k] \leftarrow L[ i ]$
8.               $i \leftarrow i + 1$
9.          else $A[k] \leftarrow R[ j ]$
10.              $j \leftarrow j + 1$
```



Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences $\boldsymbol{A}$ and $\boldsymbol{B}$ into a sorted sequence $\boldsymbol{S}$ containing the union of the elements of $\boldsymbol{A}$ and $\boldsymbol{B}$
- Merging two sorted sequences, each with $\boldsymbol{n}/2$ elements and implemented by means of a doubly linked list, takes $\boldsymbol{O}(\boldsymbol{n})$ time

Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $\boldsymbol{n}/2$ elements and implemented by means of a doubly linked list, takes at most $\boldsymbol{bn}$ steps, for some constant $\boldsymbol{b}$.
- Likewise, the basis case ($\boldsymbol{n} < 2$) will take at $\boldsymbol{b}$ most steps.
- Therefore, if we let $\boldsymbol{T(n)}$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $\boldsymbol{T(n)}$ only on the left-hand side.

Iterative Substitution

- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/2^2)) + b(n/2)) + bn \\ &= 2^2 T(n/2^2) + 2bn \\ &= 2^3 T(n/2^3) + 3bn \\ &= 2^4 T(n/2^4) + 4bn \\ &= \ldots \\ &= 2^i T(n/2^i) + ibn \end{aligned}$$

- Note that base, T(n)=b, case occurs when 2^i=n. That is, i = log n.
- So,

$$T(n) = bn + bn\log n$$

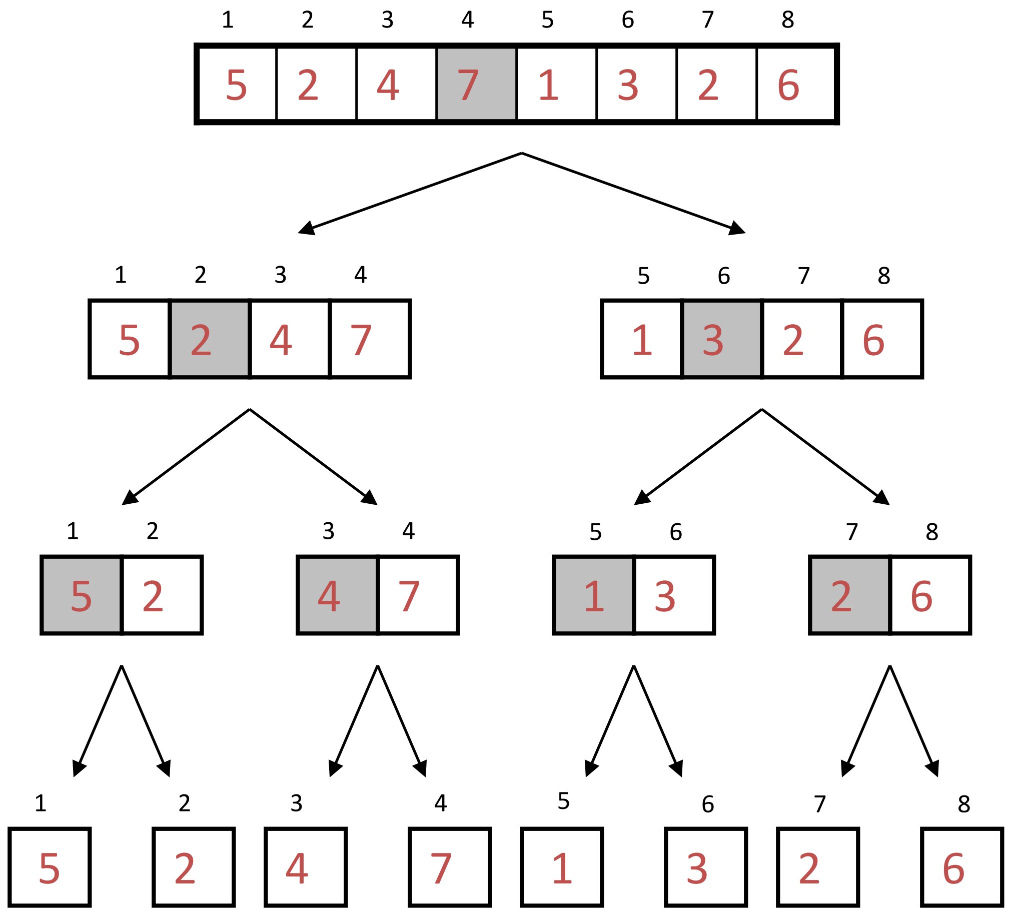
- Thus, T(n) is **O(n log n)**.

WORKING OF MERGE SORT

Example – *n* Power of 2

Divide

q = 4

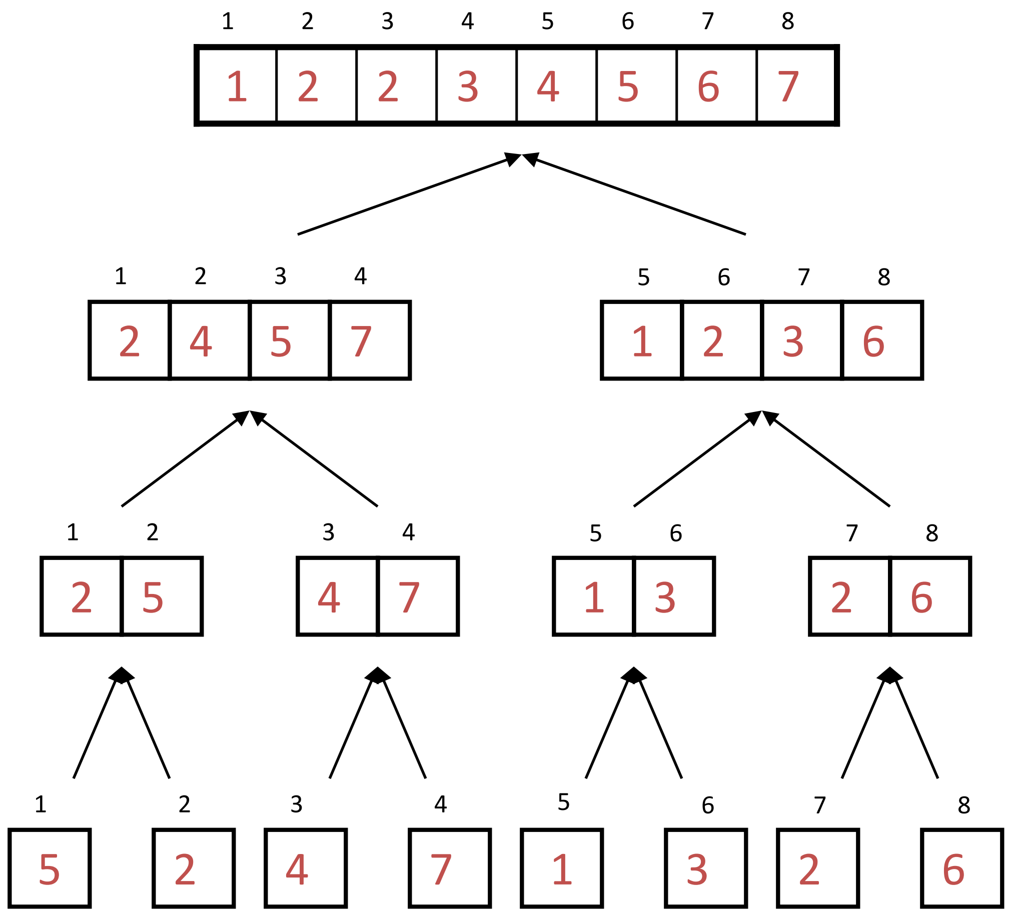


Example – *n* Power of 2

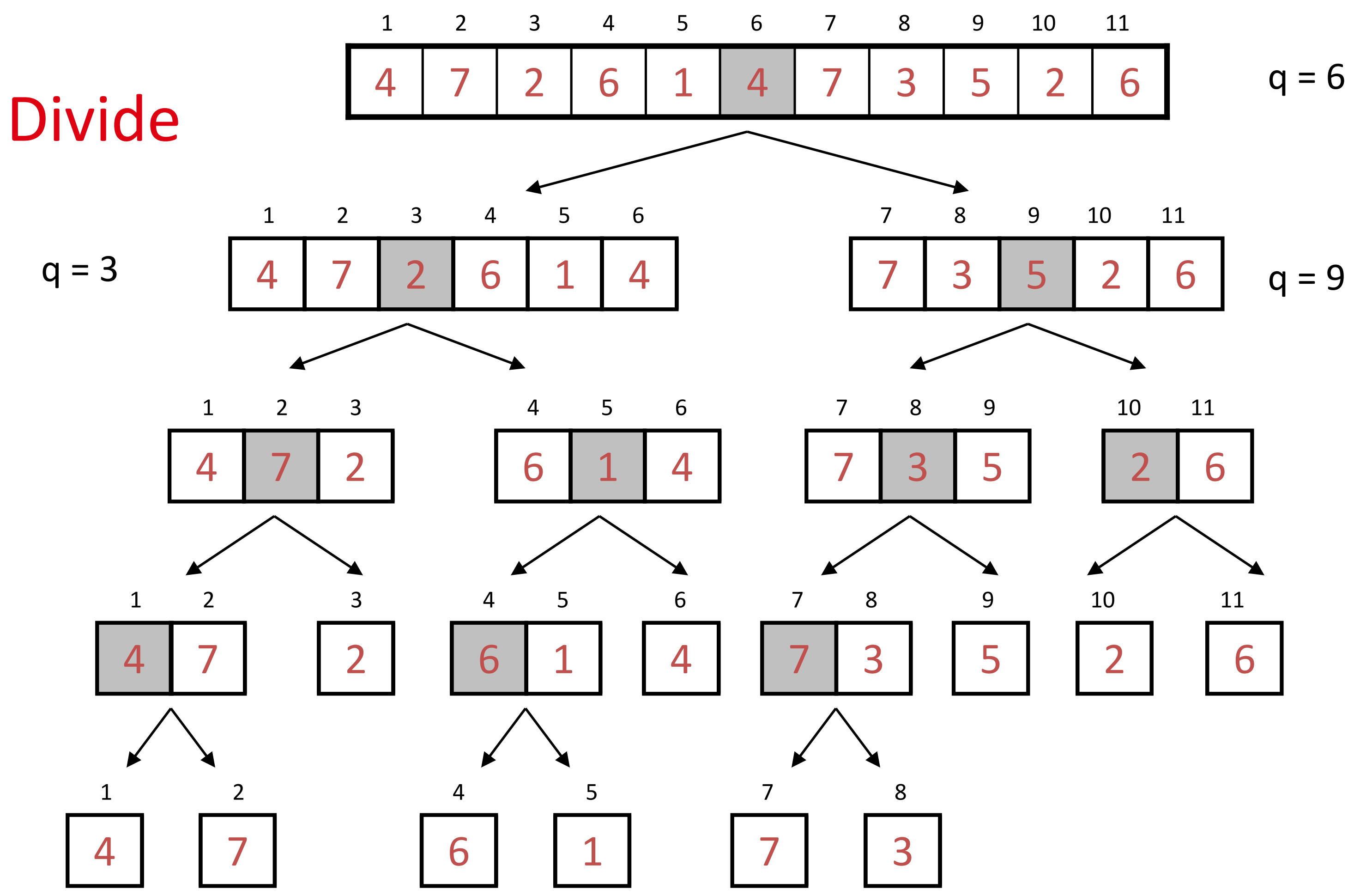
Conquer

and

Merge

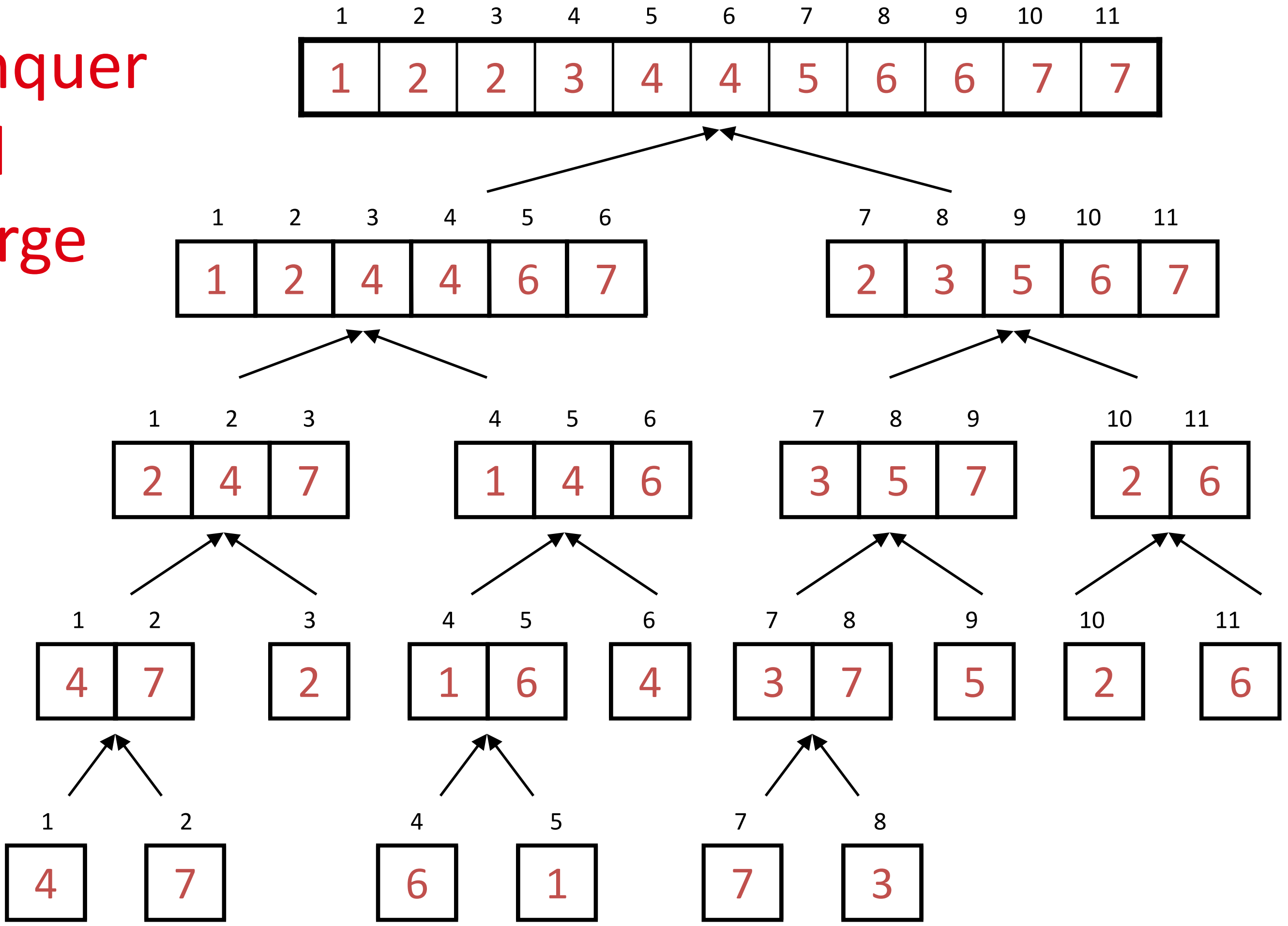


Example – *n* Not a Power of 2



Example – *n* Not a Power of 2

Conquer and Merge



Complexity of MergeSort

Pass Number	Number of merges	Merge list length	# of comps / moves per merge
1	2^{k-1} or $n/2$	1 or $n/2^k$	$\leq 2^1$
2	2^{k-2} or $n/4$	2 or $n/2^{k-1}$	$\leq 2^2$
3	2^{k-3} or $n/8$	4 or $n/2^{k-2}$	$\leq 2^3$
.	.	.	.
.	.	.	.
.	.	.	.
$k - 1$	2^1 or $n/2^{k-1}$	2^{k-2} or $n/4$	$\leq 2^{k-1}$
k	2^0 or $n/2^k$	2^{k-1} or $n/2$	$\leq 2^k$

$k = \log n$

Complexity of MergeSort

Multiplying **the number of merges** by the **maximum number of comparisons** per merge, we get:

$$(2^{k-1})2^{1} = 2^{k}$$
$$(2^{k-2})2^{2} = 2^{k}$$
$$\vdots$$
$$(2^{1})2^{k-1} = 2^{k}$$
$$(2^{0})2^{k} = 2^{k}$$

***k* passes each require *2^k* comparisons (and moves). But *k = lg n* and hence, we get lg(n) • n comparisons or O(n lgn)**

Variants of merge sort

Insertion Sort

- Insertion Sort Insertion sort inserts each item into its proper place in the final list.
- It consumes one input at a time and growing a sorted list.
- It is highly efficient on small data and is very simple and easy to implement.
- The worst case complexity of insertion sort is $O(n^2)$. When data is already sorted it is the best case of insertion sort and it takes $O(n)$ time

Insertion Sort

```
Algorithm InsertionSort($a, n$)
// Sort the array $a[1:n]$ into nondecreasing order, $n \geq 1$.
{
    for $j := 2$ to $n$ do
    {
        // $a[1:j-1]$ is already sorted.
        $item := a[j]$; $i := j-1$;
        while (($i \geq 1$) and ($item < a[i]$)) do
        {
            $a[i+1] := a[i]$; $i := i-1$;
        }
        $a[i+1] := item$;
    }
}
```

Time complexity of Insertion Sort

- The complexity of the algorithm is result of the statements within the domain of the while loop.
- This loop can executed minimum zero times to a maximum of j times. The J varies from 2 to n, hence worst case time complexity of the insertion sort can be computed as :

$$\sum_{2\le j\le n} j = \frac{n(n+1)}{2} - 1 = \Theta(n^2)$$

- If the data is already sorted, the body of the while loop is never entered, hence the best case computing time is Θ(n)

Revised version of merge sort with Insertion sort with elements are stored in linked list

Modified Merge Sort: MergSort1

```
Algorithm MergeSort1($low, high$)
// The global array $a[low : high]$ is sorted in nondecreasing order
// using the auxiliary array $link[low : high]$. The values in $link$
// represent a list of the indices $low$ through $high$ giving $a[\,]$ in
// sorted order. A pointer to the beginning of the list is returned.
{
    if $((high - low) < 15)$ then
        return InsertionSort1($a, link, low, high$);
    else
    {
        $mid := \lfloor (low + high)/2 \rfloor$;
        $q :=$ MergeSort1($low, mid$);
        $r :=$ MergeSort1($mid + 1, high$);
        return Merge1($q, r$);
    }
}
```

Merging link list of sorted elements

```
Algorithm Merge1$(q, r)$
// $q$ and $r$ are pointers to lists contained in the global array
// $link[0:n]$. $link[0]$ is introduced only for convenience and need
// not be initialized. The lists pointed at by $q$ and $r$ are merged
// and a pointer to the beginning of the merged list is returned.
{
    $i := q$; $j := r$; $k := 0$;
    // The new list starts at $link[0]$.
    while $((i \neq 0)$ and $(j \neq 0))$ do
    { // While both lists are nonempty do
        if $(a[i] \leq a[j])$ then
        { // Find the smaller key.
            $link[k] := i$; $k := i$; $i := link[i]$;
            // Add a new key to the list.
        }
        else
        {
            $link[k] := j$; $k := j$; $j := link[j]$;
        }
    }
    if $(i = 0)$ then $link[k] := j$;
    else $link[k] := i$;
    return $link[0]$;
}
```

Merge sort on linked list

100

ADDR	DATA	LINK
100	11	108
108	3	116
116	39	124
124	50	132
132	21	140
140	42	148
148	7	156
156	18	164
164	6	172
172	36	NULL

108

ADDR	DATA	LINK
100	11	156
108	3	164
116	39	140
124	50	NULL
132	21	172
140	42	124
148	7	100
156	18	132
164	6	148
172	36	116

92

ADDR	DATA	LINK
92		100
100	11	108
108	3	116
116	39	124
124	50	132
132	21	140
140	42	148
148	7	156
156	18	164
164	6	172
172	36	NULL

92

ADDR	DATA	LINK
92		108
100	11	156
108	3	164
116	39	140
124	50	NULL
132	21	172
140	42	124
148	7	100
156	18	132
164	6	148
172	36	116

Sorting Challenge 1

Problem: Sort a file of huge records with tiny keys

Example application: Reorganize your MP-3 files

Which method to use?

A. merge sort, guaranteed to run in time O(Nlog)N
B. selection sort
C. bubble sort
D. a custom algorithm for huge records/tiny keys
E. insertion sort

Sorting Files with Huge Records and Small Keys

- Insertion sort or bubble sort?
  - NO, too many exchanges
- Selection sort?
  - YES, it takes linear time for exchanges
- Merge sort or custom method?
  - Probably not: selection sort simpler, does less swaps

Sorting Challenge 2

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. Bubble sort
B. Selection sort
C. Mergesort guaranteed to run in time O(NlogN)
D. Insertion sort

Sorting Huge, Randomly - Ordered Files

- Selection sort ?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Mergesort?
  - YES, it is designed for this problem

Sorting Challenge 3

Problem: sort a file that is already almost in order

Applications:

- Re-sort a huge database after a few changes
- Doublecheck that someone else sorted a file

Which sorting method to use?

A. Mergesort, guaranteed to run in time ~NlgN

B. Selection sort

C. Bubble sort

D. A custom algorithm for almost in-order files

E. Insertion sort

Sorting Files That are Almost in Order

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, bad for some definitions of “almost in order”
  - Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Insertion sort?
  - YES, takes linear time for most definitions of “almost in order”
- Mergesort or custom method?
  - Probably not: insertion sort simpler and faster

Exercises

Chapter 3 : Divide & Conqure

4. Suppose $a[1:m]$ and $b[1:n]$ both contain sorted elements in non-decreasing order. Write an algorithm that merges these items into $c[1:m+n]$. Your algorithm should be shorter than Algorithm 3.8 (Merge) since you can now place a large value in $a[m+1]$ and $b[n+1]$.

5. Given a file of n records that are partially sorted as $x_1 \leq x_2 \leq \cdots \leq x_m$ and $x_{m+1} \leq \cdots \leq x_n$, is it possible to sort the entire file in time $O(n)$ using only a small fixed amount of additional storage?

6. Another way to sort a file of n records is to scan the file, merge consecutive pairs of size one, then merge pairs of size two, and so on. Write an algorithm that carries out this process. Show how your algorithm works on the data set (100, 300, 150, 450, 250, 350, 200, 400, 500).

7. A version of insertion sort is used by Algorithm 3.10 to sort small subarrays. However, its parameters and intent are slightly different from the procedure InsertionSort of Algorithm 3.9. Write a version of insertion sort that will work as Algorithm 3.10 expects.

8. The sequences $X_1, X_2, \ldots, X_\ell$ are sorted sequences such that $\sum_{i=1}^{\ell} |X_i| = n$. Show how to merge these ℓ sequences in time $O(n \log \ell)$.

Suggested Exercises

1. Is merge sort a stable sorting algorithm?
2. How would you implement merge sort without using recursion?
3. Determine the running time of merge sort for (i) sorted input, (ii) reverse-ordered input, and (iii) random input.
4. The worst case running time of mergesort is O(n logn). What is its best-case time? Can we say that the time for merge sort is θ(n log n)?
5. Consider the following variation on MERGESORT for large values of n. Instead of recursing until n is sufficiently small recur at most a constant r times, and then use insertion sort to solve the 2r resulting sub-problems. What is the (asymptotic) running time of this variation as a function of n?

Suggested Exercises

6. Can we make a general statement that "if a problem can be split using D&C strategy in almost equal portion at each stage, then it is a good candidate for recursive implementation, but if it cannot be easily divided into equal portions, than it is better to be implemented iteratively“? Explain with the help of an example.

7. It is suggested that MERGE SORT is not a good strategy for arrays of size less than about 8 to 16, for which "INSERTION SORT" may be better. Realizing that in our MERGE SORT algorithm, It is the arrays of small sizes which are handled very often, it may be a good idea to modify our MERGE algorithm so that when the size of an sub-array is less than, say 8, it is not split further, but is sorted by INSERTION SORT.

Suggested Exercise

8. Derive the time complexity of the modified merge sort algorithm with above modification, assuming that insertion sort takes a fixed time b.

Speak 5 lines to YOURSELF Every Morning:

1. I am the best.
2. I can do it.
3. God is always with me.
4. I am a winner.
5. Today is my day.

-Dr. A.P.J. Abdul Kalam

Thanks for Your Attention!