BML MUNJAL UNIVERSITY™

Data Structures & Algorithms

Faster sorting algorithms

Merging two sorted arrays

Precondition :

Let A is a list of size m, and B is a list of size n, m, n >0.

Both the lists A and B are sorted.

Post condition :

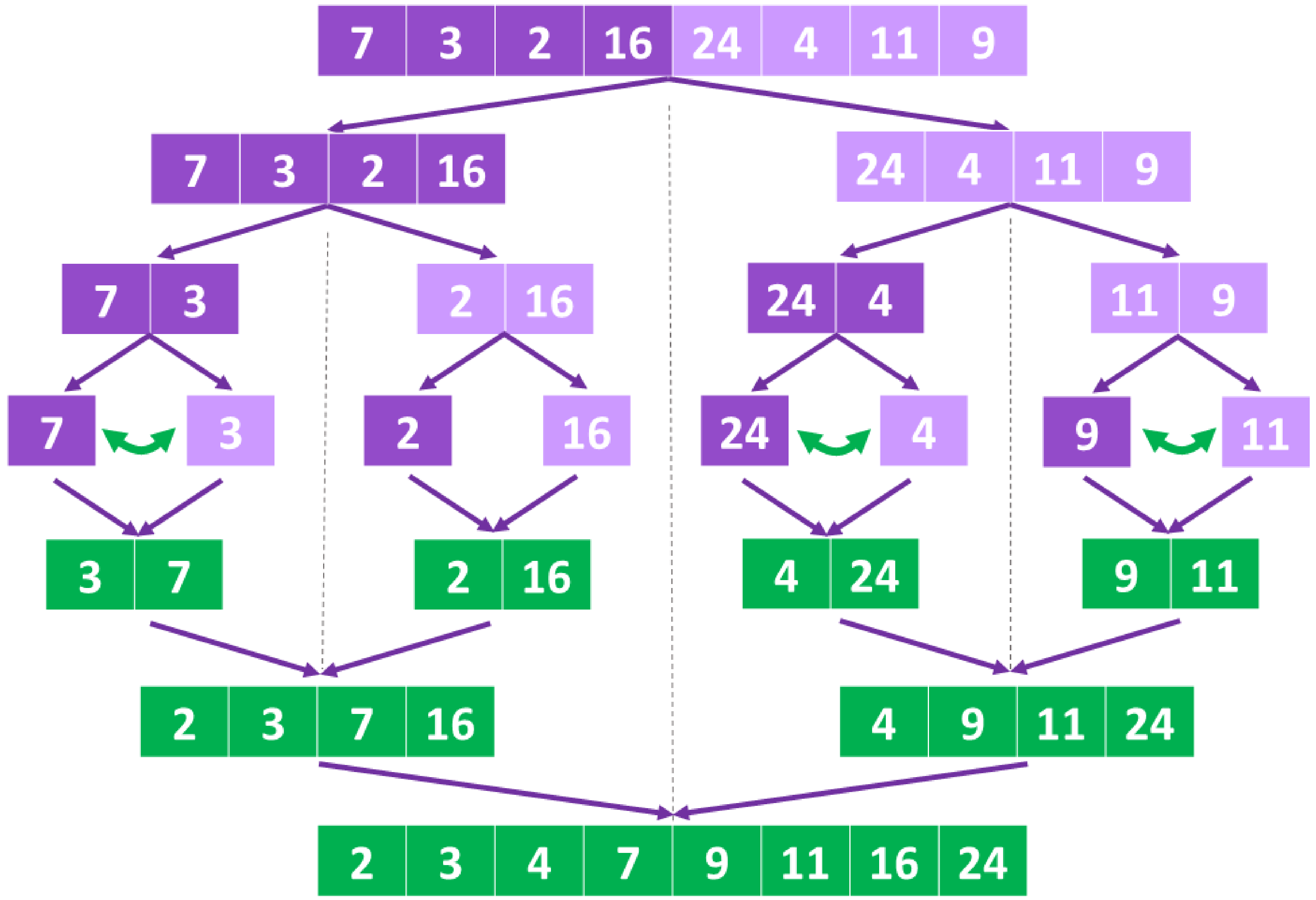
Combined A and B to make a larger sorted list C of size m+n.

Write an algorithm to merge two sorted lists of length m and n respectively, into a single sorted list of size m+n.

Merging two sorted arrays

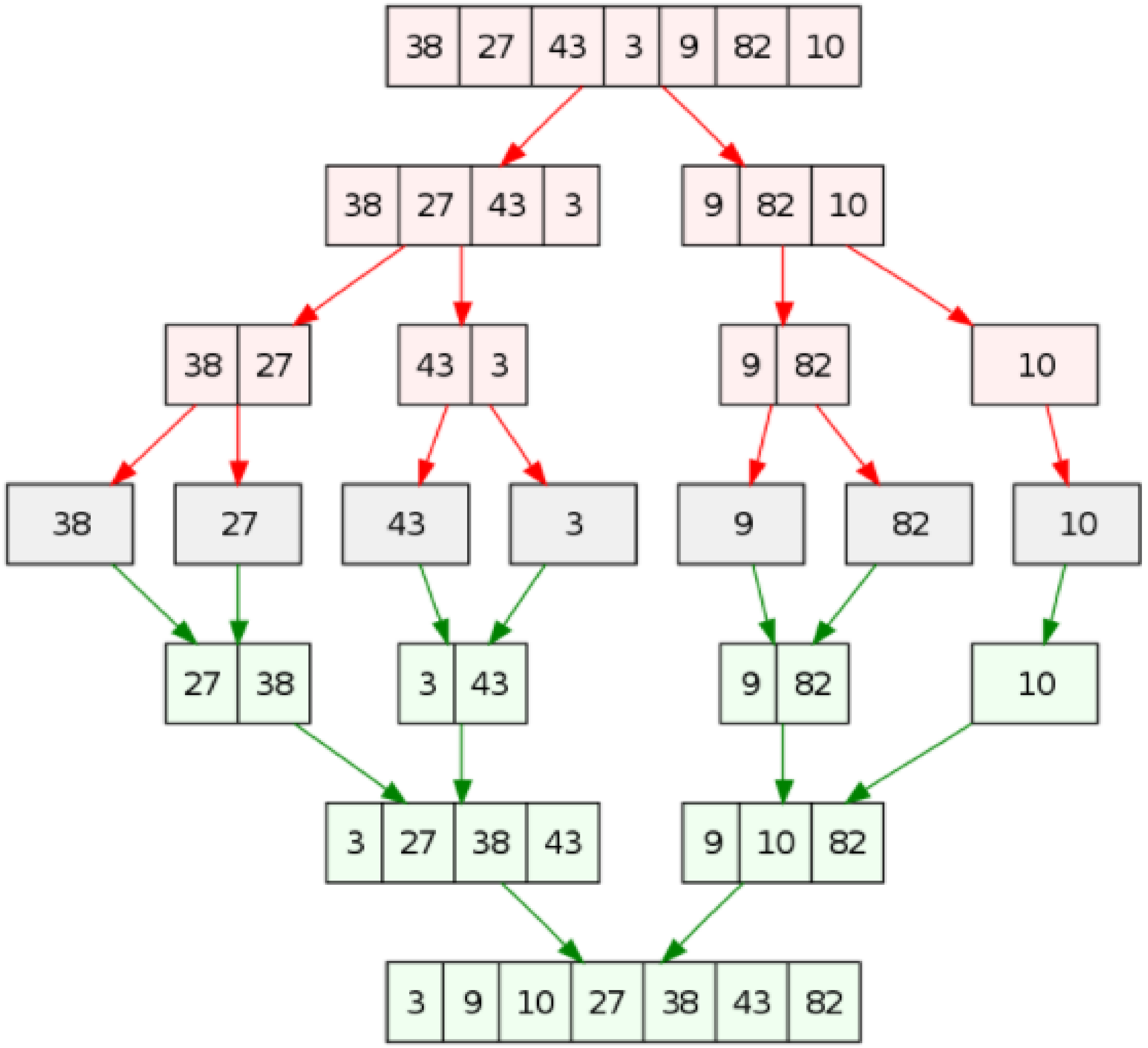
```
Merge(A[], B[], m, n)
{
  i=0; j=0; k=0;
  while (i<m and j<n)  {
     if (A[i]<=B[j])
         C[k]=A[i];    i=i+1;
     else
          C[k]=B[j];  j=j+1;
     k=k+1;
  }
  while (i<m) {
     C[k]=A[i];
     i=i+1;  k=k+1;
   }
   while (j<n) {
     C[k]=B[j];
     j=j+1;  k=k+1;
   }
   return C;
}
```

Merge Sort



7 3 2 16 24 4 11 9
7 3 2 16
24 4 11 9
7 3
2 16
24 4
11 9
7
3
2
16
24
4
9
11
3 7
2 16
4 24
9 11
2 3 7 16
4 9 11 24
2 3 4 7 9 11 16 24

Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merge Sort

Mergesort is a fast sorting algorithm.

Idea behind merge sort :

Divide/Split array down the middle into two halves, each of size roughly n/2.

Sort the left half of the elements (recursively).

Sort the right half of the elements (recursively).

Merge the two sorted halves into a single sorted array.

Merge Sort

```
MergeSort(A[], left, right)
{
    if (left < right)  {            // when we have more than one elements
        mid = (left + right)/2
        MergeSort(A, left, mid) // Recursively sort the left half
        MergeSort(A, mid+1, right) // Recursively sort the right half
        Merge(A, left, mid, right) // merge the two sorted halves
    }
    else
       return;
}
```

Complexity analysis of Merge Sort

Time complexity

O(nlogn) in best, worst and average cases.

Disadvantage/drawback

Merging phase requires an extra array, more importantly, the additional work spent copying to the temporary array and back slow down the sort process. So, Merge sort requires O(n) extra space, including O(log(n)) stack space for the recursive calls.

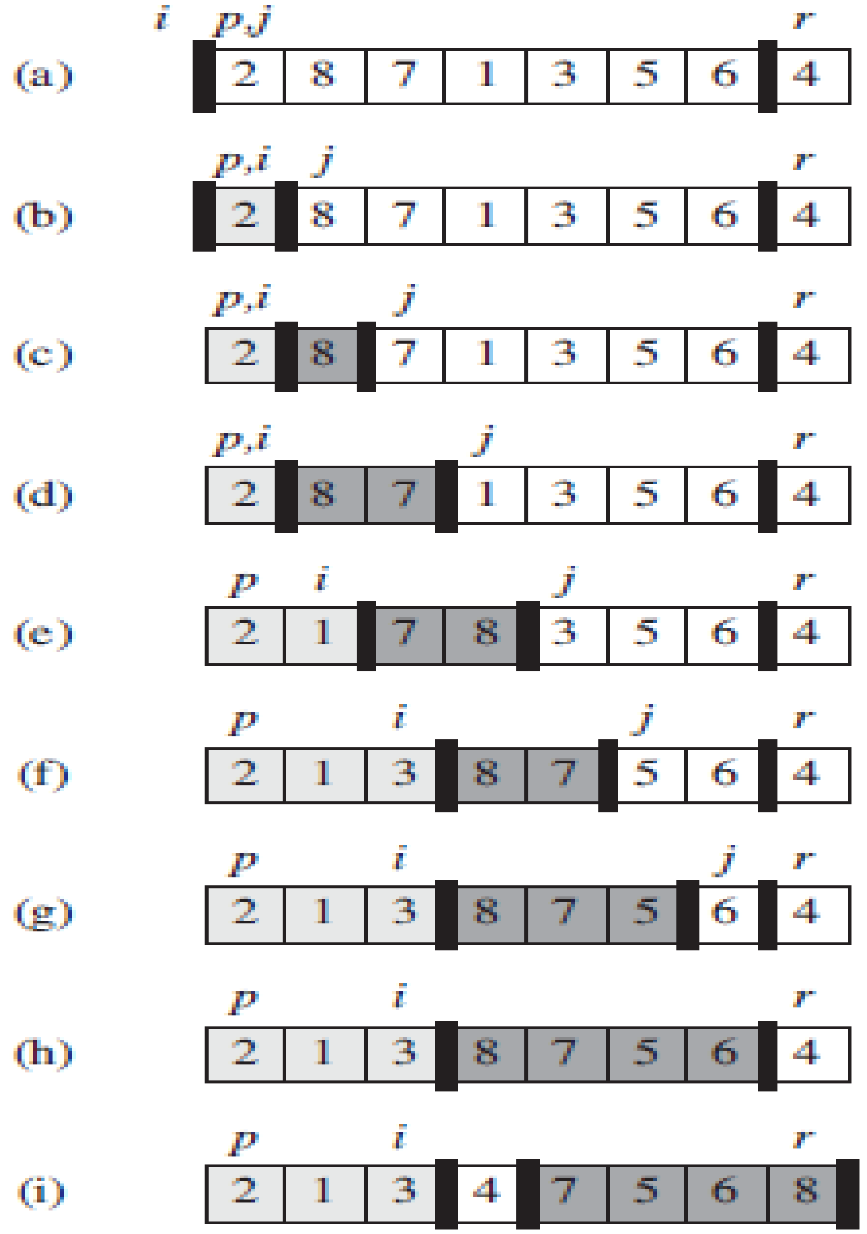
Quick Sort

Quicksort is the most popular(most commonly used) sorting algorithm, and usually is very fast.

Quicksort is complementary to mergesort.

It avoids mergesort’s problem of needing an extra array to store the final sorted output.

Quick Sort



(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8

Quick Sort(deterministic)

Basic idea behind quick sort :

The classic quicksort algorithm to sort an array A consists of the following four easy steps:

1. If the number of elements in A is less than 2, then return. Otherwise(when n>1),

2. Pick/choose/select any element e in A as a pivot.

[pivot is an element with respect to whose value we are going to divide the subarray]

3. Partition A − {e} (the remaining elements in A) into two disjoint groups: $A1 = \{x \in A - \{e\} \mid x \leq e\}$, and $A2 = \{x \in A - \{e\} \mid x \geq e\}$.

4. Recursively Call {quicksort(A1) followed by e followed by quicksort(A2)}.

After every partitioning, the pivot is in its final position.

Quick Sort

```
QuickSort(A[], left, right)
{
// In the initial call,  left=0, right=n − 1, defined by its left and right indices
   if (left < right)   {
     piv =Partition(A, left, right)
     Quicksort(A, left, piv − 1)
     Quicksort(A, piv + 1, right)
 }
```

Quick Sort

The key to the algorithm is the partition algorithm, which rearranges the subarray A[left…right] in place.

```
Partition(A[], left, right)
{
   piv = A[right];  // we choose the last element of the array as pivot
   i= left – 1;
   for (j = left; j<=right – 1; j++)   {
      if (A[j] ≤ piv)
          i= i+ 1;
          if (i != j)
              swap(A[i], A[j]);
   }
   swap(A[i+ 1], A[right]);
   return (i + 1);
}
```

Complexity analysis of Quick Sort

Time complexity

O(nlogn) in best and average cases.

O(n^2) in worst-case.

It avoids Mergesort’s problem of needing an extra array during the merge operation.

Quicksort takes O(log n) stack space for the recursive calls.

Can we have O(n) worst case time sorting algorithm?

Comparison-based sorts can't be faster than nlogn. BUT non-comparison based ones can.

Sorting : The Big Picture

Simple algorithm : Selection sort, Bubble sort, Insertion sort

Sophisticated algorithm : Merge sort, Heap sort, Quick sort(average case)

Non comparison based algorithm : Counting sort, Radix sort, Bucket sort