Merge Sort

CSE 2011

Fall 2009

Goals

- Divide-and-conquer approach
- Solving recurrences
- One more sorting algorithm

Merge Sort: Main Idea

Based on divide-and-conquer strategy

- Divide the list into two smaller lists of about equal sizes.
- Sort each smaller list *recursively.*
- Merge the two sorted lists to get one sorted list.

How do we divide the list? How much time needed?

How do we merge the two sorted lists? How much time needed?

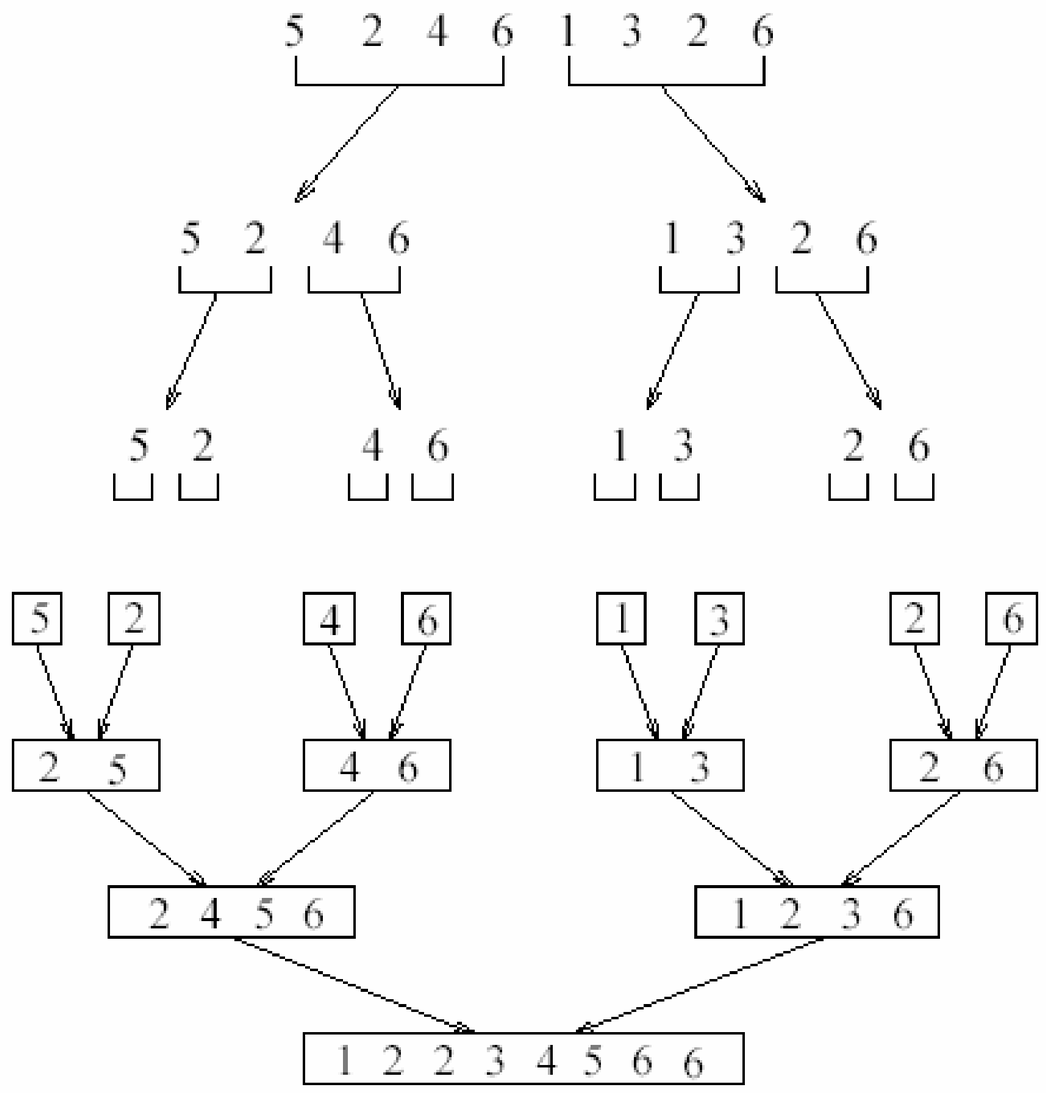
Dividing

- If the input list is a linked list, dividing takes $\Theta(N)$ time:
  - Scan the linked list, stop at the $\lfloor N/2 \rfloor^{th}$ entry and cut the link.
- If the input list is an array A[0..N-1]: dividing takes O(1) time:
  - Represent a sub-array by two integers *left* and *right*.
  - To divide A[*left* .. *right*], compute *center*=(*left*+*right*)/2 and obtain A[*left* .. *center*] and A[*center*+1 .. *right*]

Merge Sort: Algorithm

- Divide-and-conquer strategy
  - recursively sort the first half and the second half
  - merge the two sorted halves together

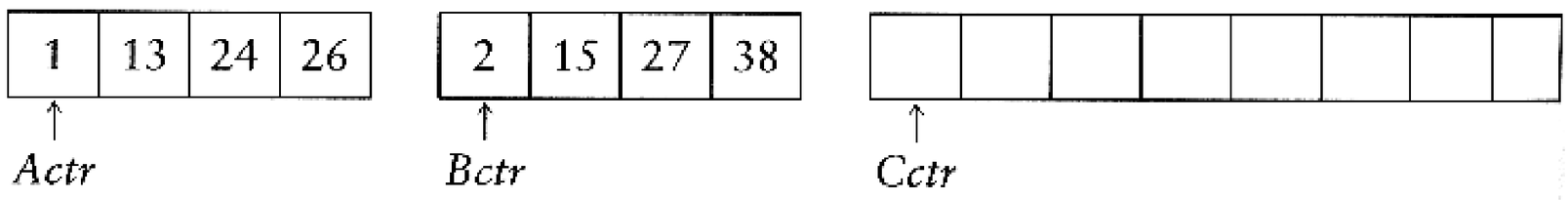
```
void mergesort(int & A[], int left, int right)
{
        If (left < right ) {
                int center = (left + right) /2;
                mergesort(A, left, center);
                mergesort(A, center+1, right);
                merge(A, left, center+1, right);
}
```

see applet http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/MSort.html

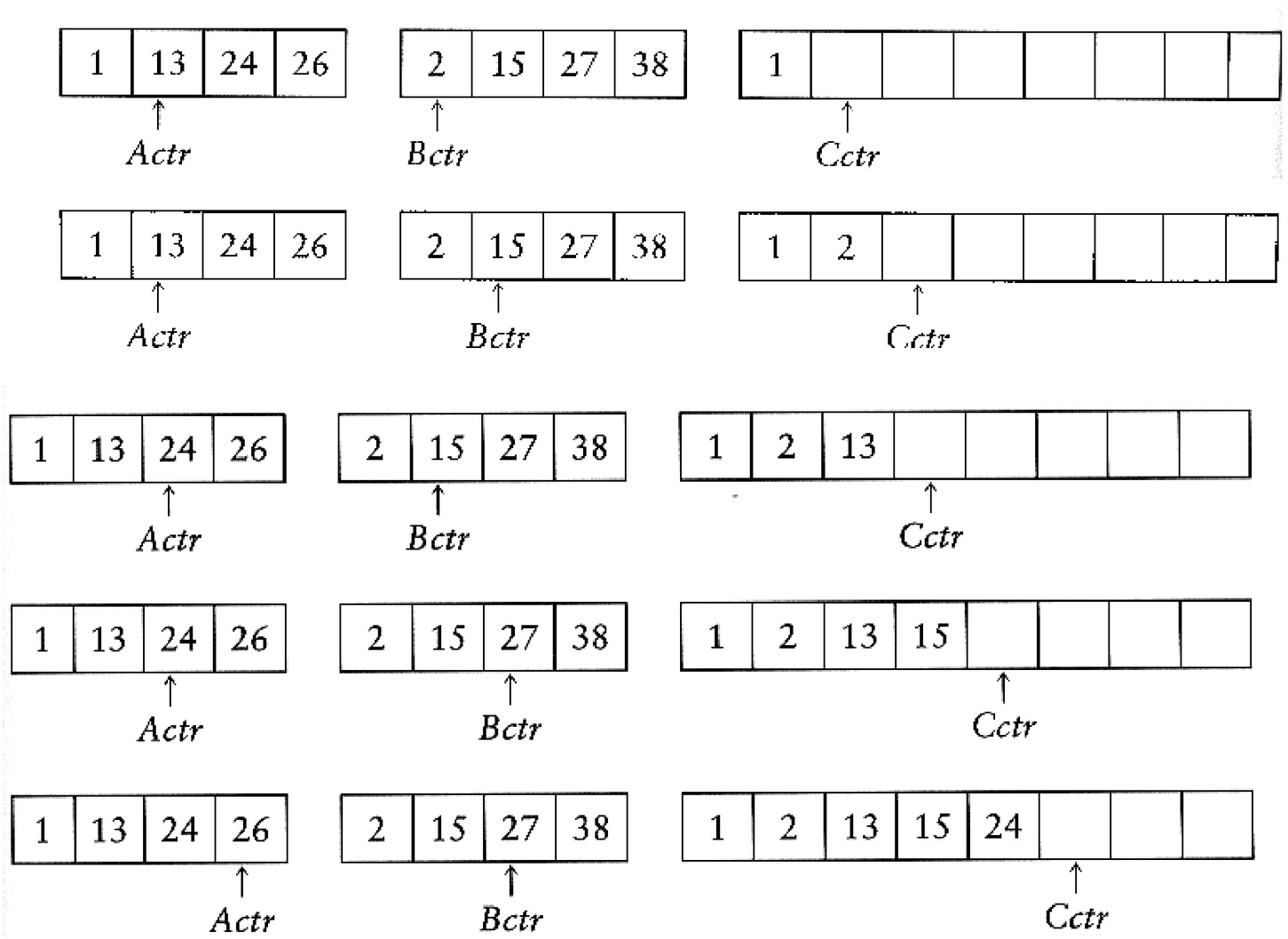
Merging

- Input: two sorted array A and B
- Output: an output sorted array C
- Three counters: *Actr*, *Bctr*, and *Cctr*
  - initially set to the beginning of their respective arrays

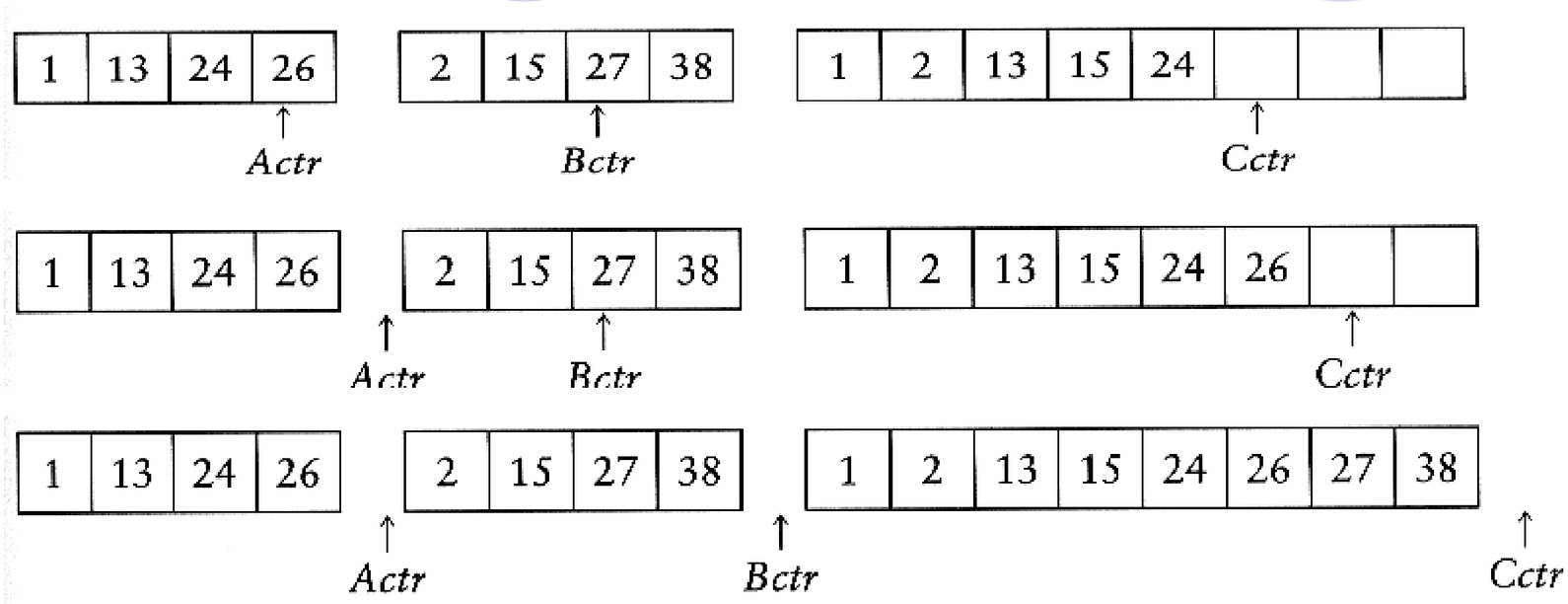


  - The smaller of A[*Actr*] and B[*Bctr*] is copied to the next entry in C, and the appropriate counters are advanced
  - When either input list is exhausted, the remainder of the other list is copied to C.

Merge: Example



Example: Merge (cont'd)



Merge Algorithm

```
/**
    * Internal method that merges two sorted halves of a
      subarray.
    * @param a an array of Comparable items.
    * @param tmpArray an array to place the merged
      result.
    * @param leftPos the left-most index of the subarray.
    * @param rightPos the index of the start of the
      second half.
    * @param rightEnd the right-most index of the
      subarray.
    */
   private static <AnyType extends Comparable<? super
      AnyType>>
   void merge( AnyType [ ] a, AnyType [ ] tmpArray, int
      leftPos, int rightPos, int rightEnd )
   {
      int leftEnd = rightPos - 1;
      int tmpPos = leftPos;
      int numElements = rightEnd - leftPos + 1;

// Main loop
      while( leftPos <= leftEnd && rightPos <= rightEnd )
         if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];
         else
            tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      while( leftPos <= leftEnd )    // Copy rest of first half
         tmpArray[ tmpPos++ ] = a[ leftPos++ ];

      while( rightPos <= rightEnd )  // Copy rest of right
      half
         tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      // Copy tmpArray back
      for( int i = 0; i < numElements; i++, rightEnd-- )
         a[ rightEnd ] = tmpArray[ rightEnd ];
   }
```

Merge: Analysis

- Running time analysis:
  - Merge takes O(m_1 + m_2), where m_1 and m_2 are the sizes of the two sub-arrays.

- Space requirement:
  - merging two sorted lists requires linear extra memory
  - additional work to copy to the temporary array and back

Analysis of Merge Sort

- 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

  T(N) = 2T(N/2) + N

Solving the Recurrence

$$T(N) = 2T(\frac{N}{2}) + N$$

$$= 2(2T(\frac{N}{4}) + \frac{N}{2}) + N$$

$$= 4T(\frac{N}{4}) + 2N$$

$$= 4(2T(\frac{N}{8}) + \frac{N}{4}) + 2N$$

$$= 8T(\frac{N}{8}) + 3N = \cdots$$

$$= 2^k T(\frac{N}{2^k}) + kN$$

Since $N=2^k$, we have $k=\log_2 n$

$$T(N) = 2^k T(\frac{N}{2^k}) + kN$$

$$= N + N\log N$$

$$= O(N\log N)$$