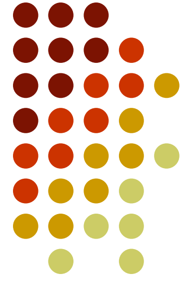
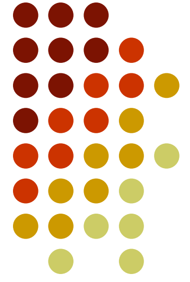


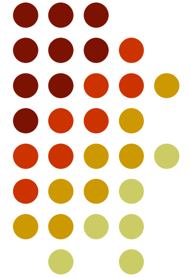
Back to Sorting – More efficient sorting algorithms

Merge Sort



- Strategy
  - break problem into smaller subproblems
  - recursively solve subproblems
  - combine solutions to answer
- Called ”***divide-and-conquer***”
  - we used the divide&conquer strategy in the binary search algorithm

Merge Sort: Algorithm

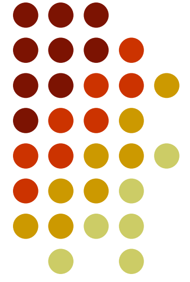


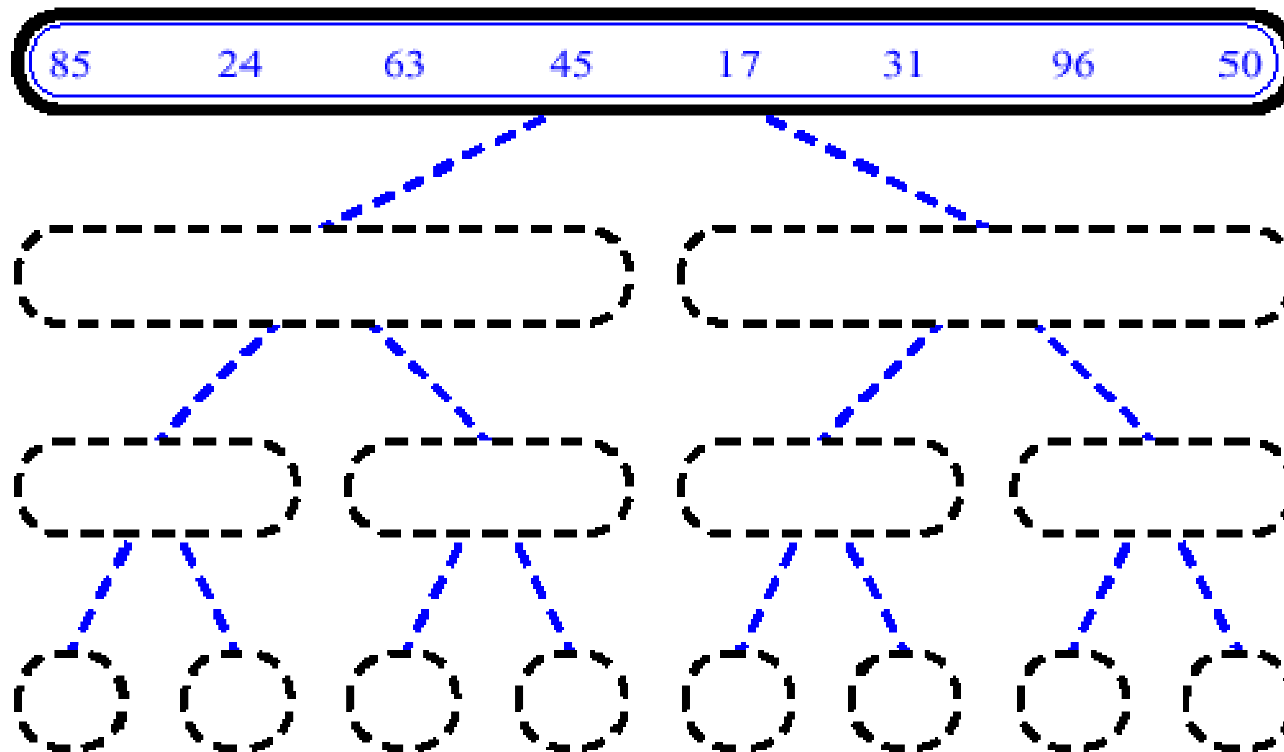
```
Merge-Sort(A, p, r)
   if p < r then
      q←⌊(p+r)/2⌋
      Merge-Sort(A, p, q)
      Merge-Sort(A, q+1, r)
      Merge(A, p, q, r)
```

Merge(A, p, q, r) – merge two ordered sequence p..q and q+1..r to one ordered sequence and store it in interval p..r.

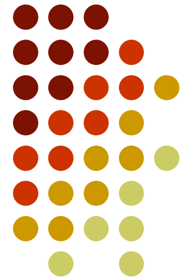
```
Merge(A, p, q, r)
   Take the smallest of the two frontmost elements of
sequences A[p..q] and A[q+1..r] and put it into a
temporary array. Repeat this, until both sequences are
empty. Copy the resulting sequence from temporary
array into A[p..r].
```

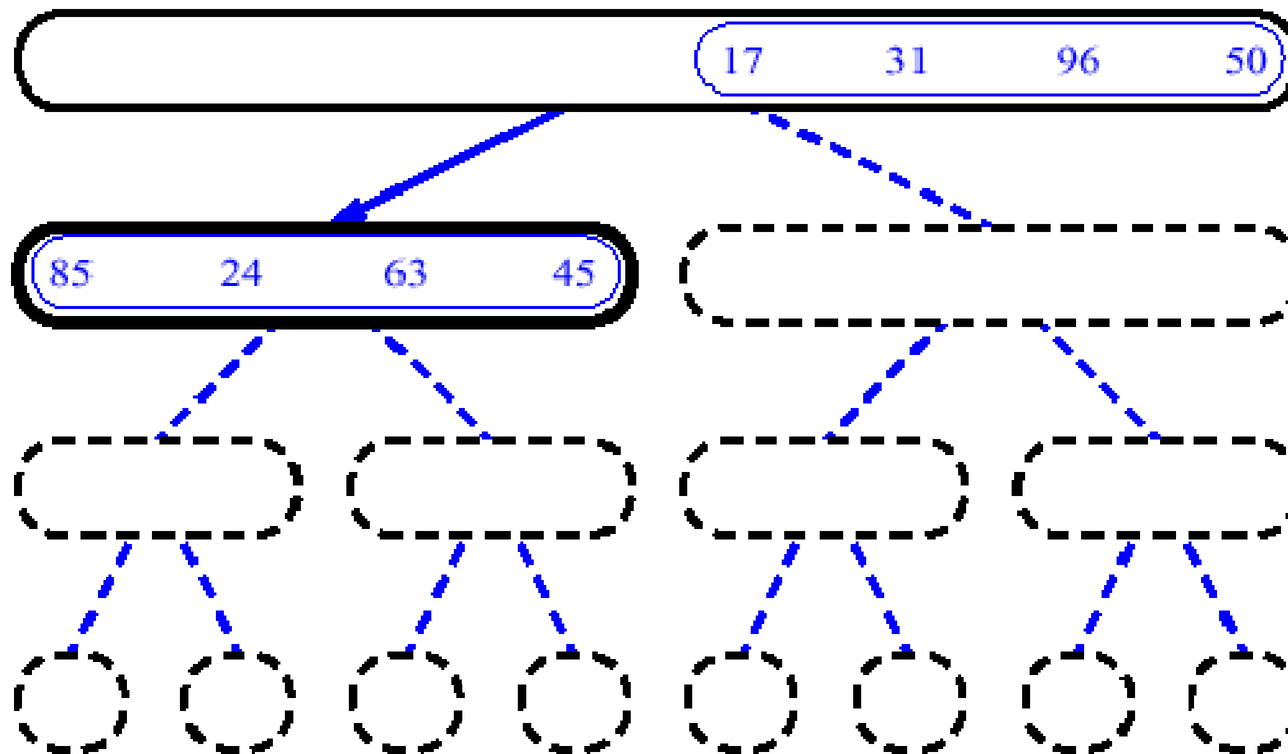
MergeSort (Example) - 1



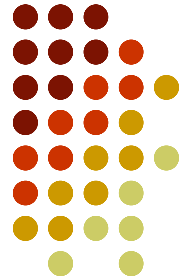


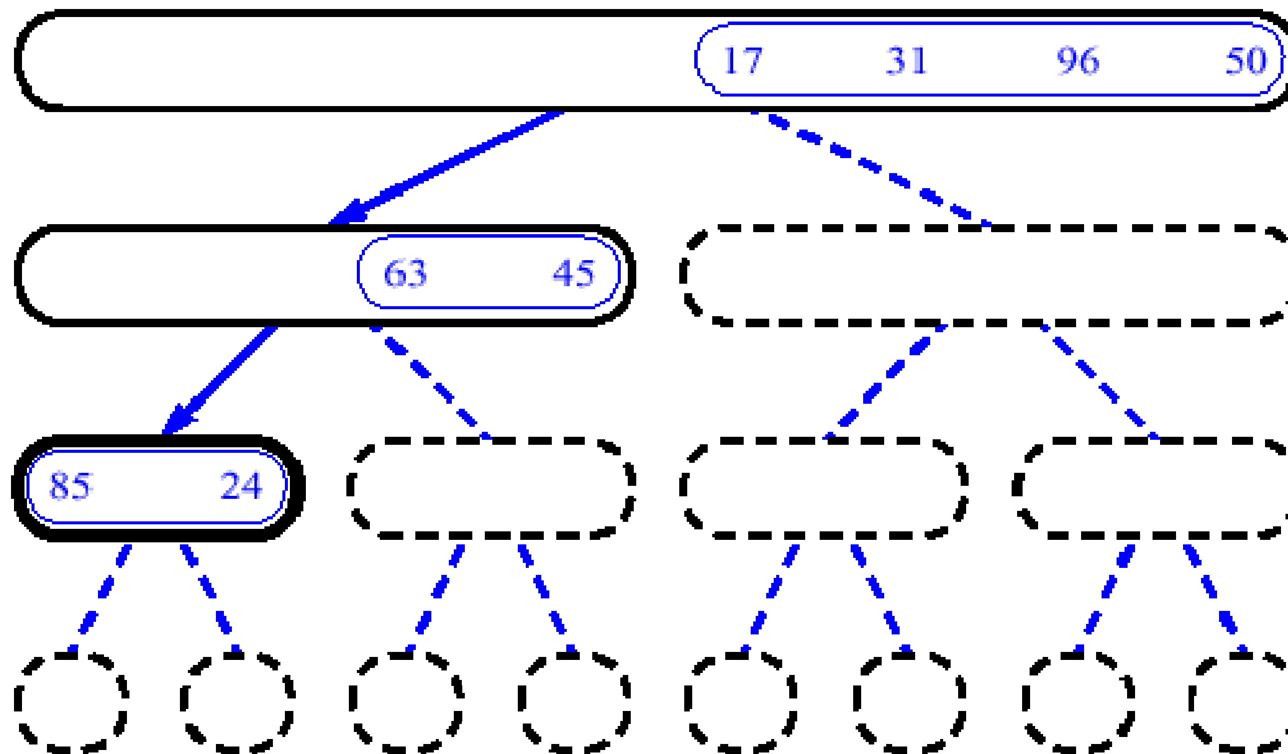
MergeSort (Example) - 2



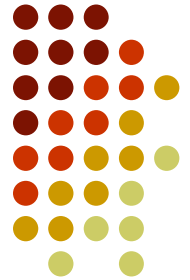


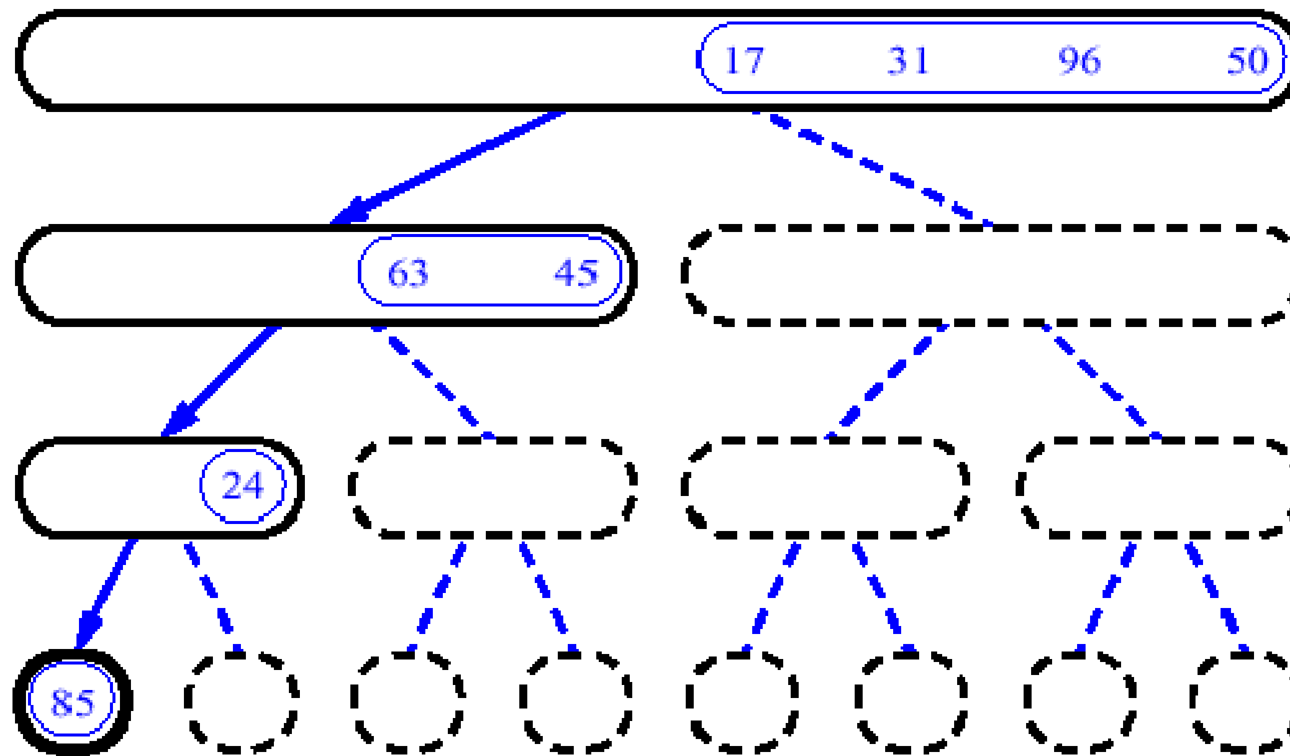
MergeSort (Example) - 3



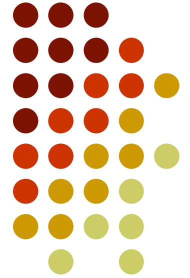


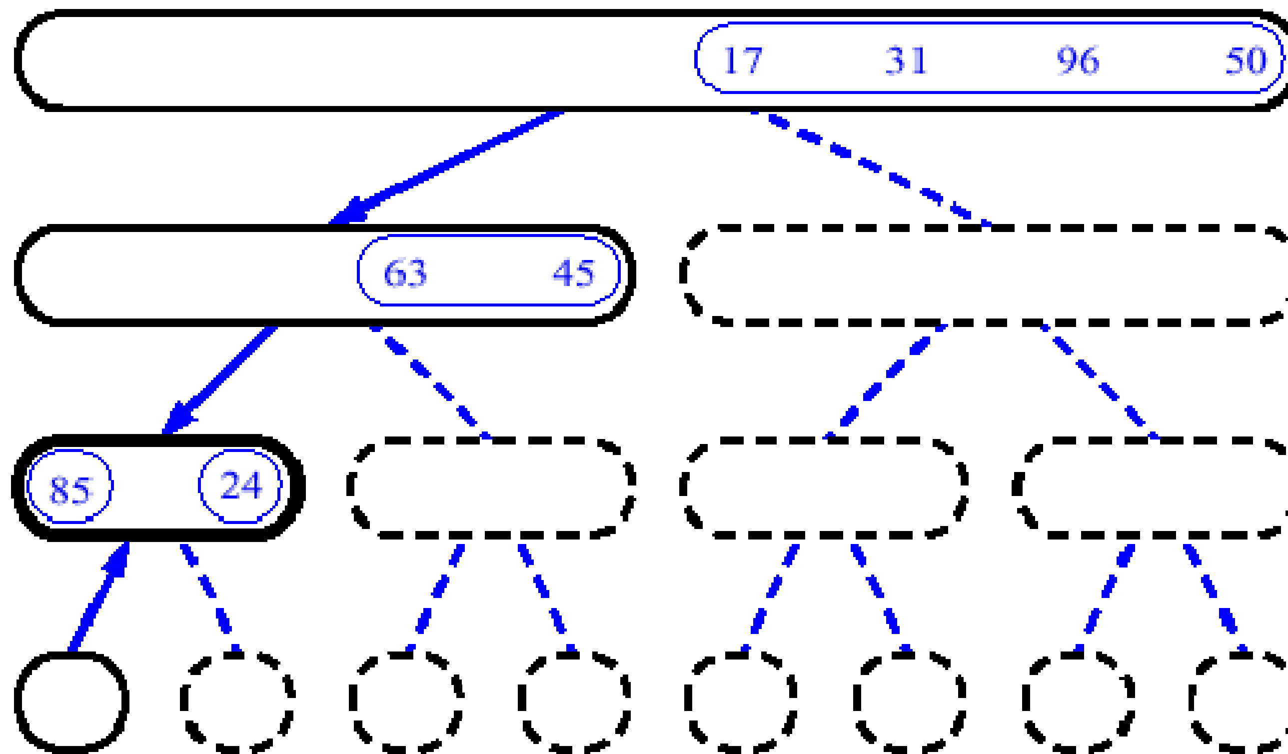
MergeSort (Example) - 4



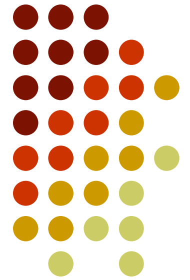


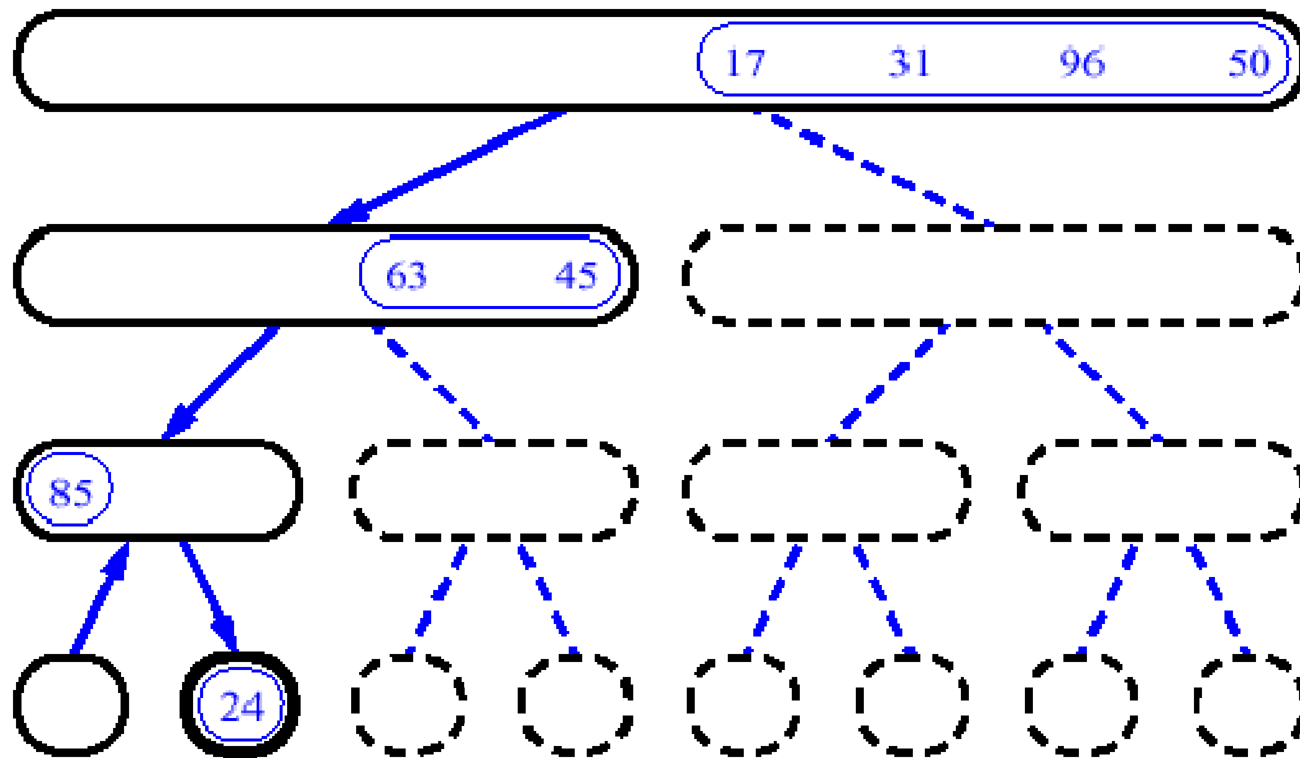
MergeSort (Example) - 5



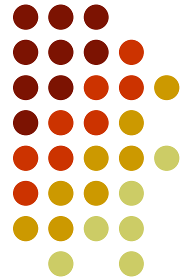


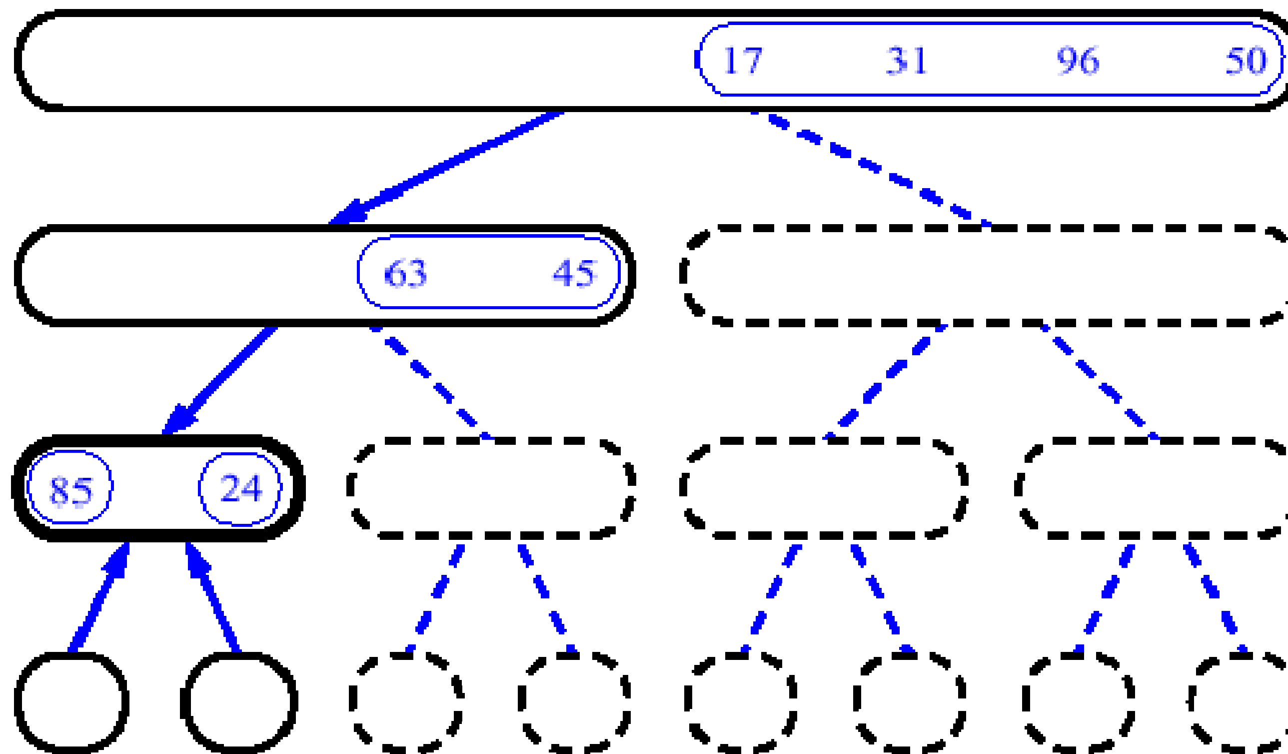
MergeSort (Example) - 6



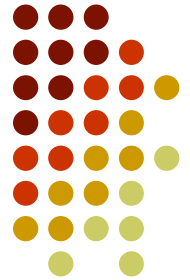


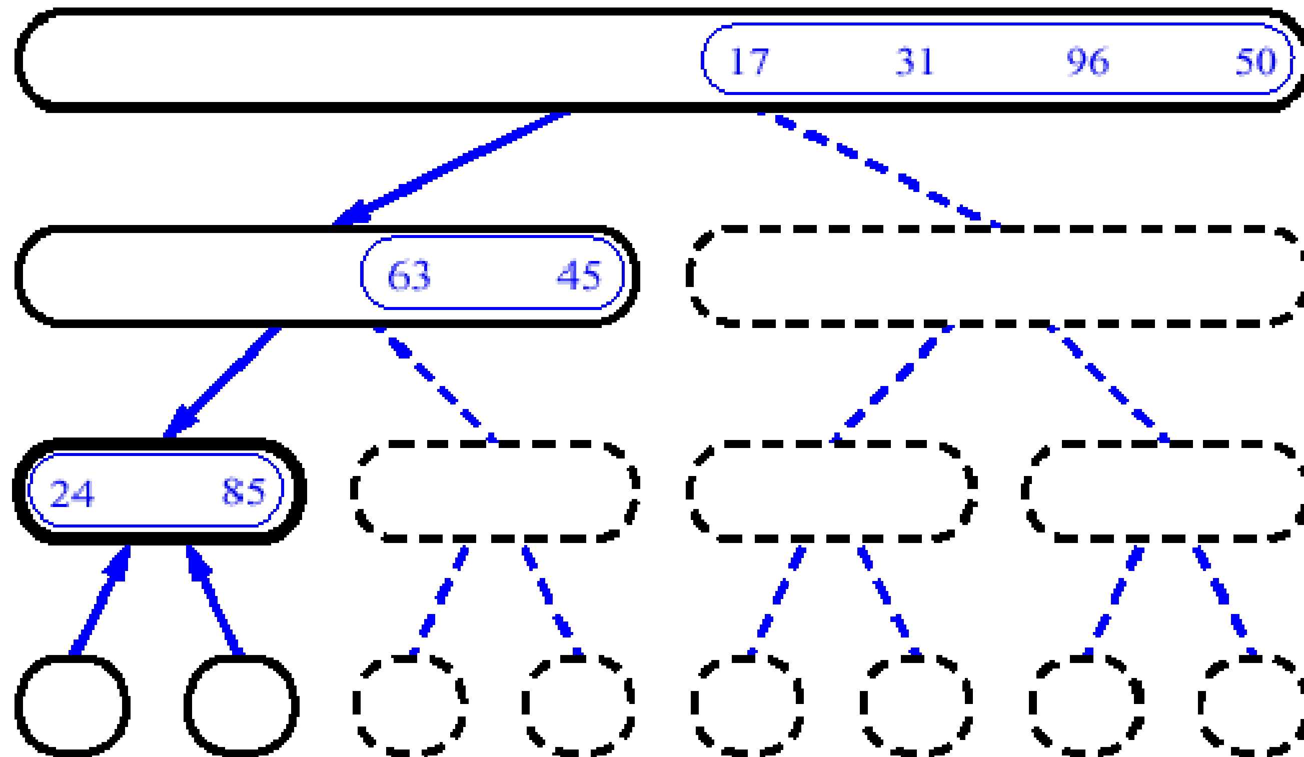
MergeSort (Example) - 7



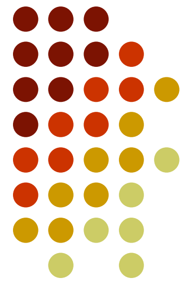


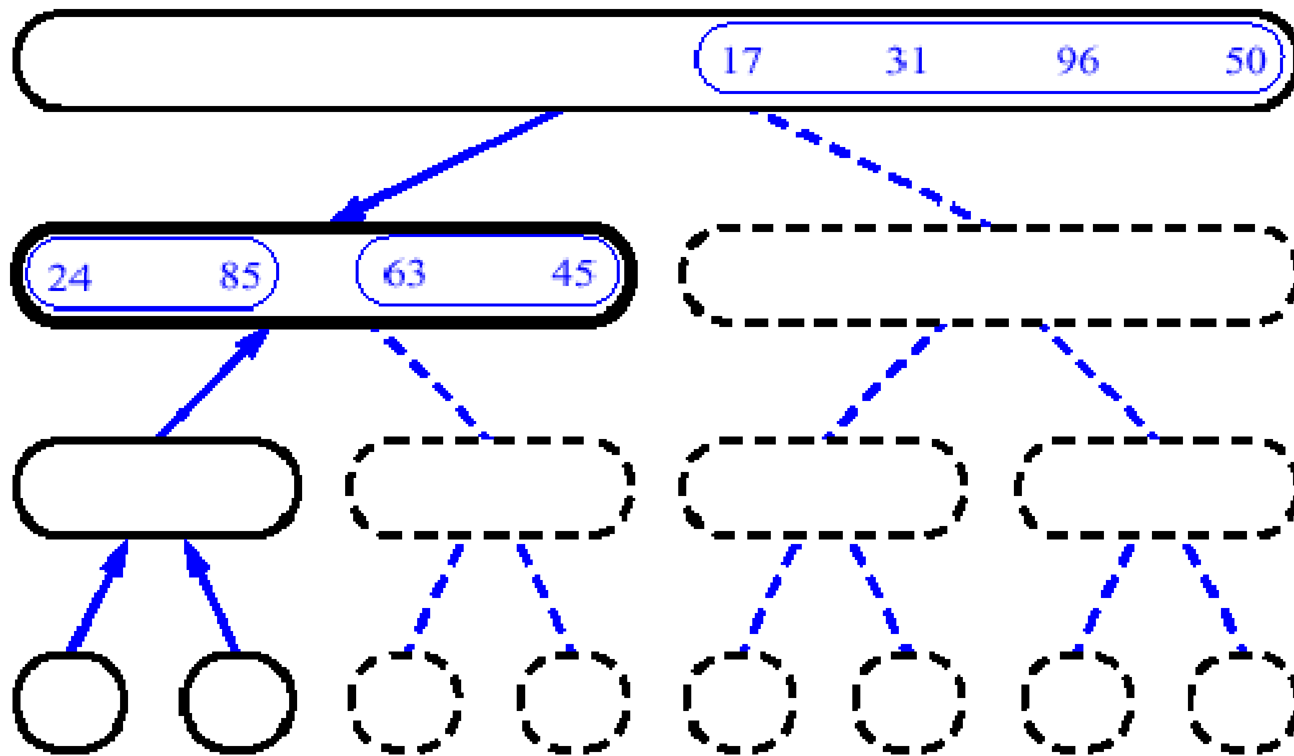
MergeSort (Example) - 8



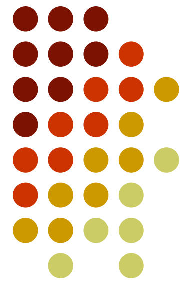


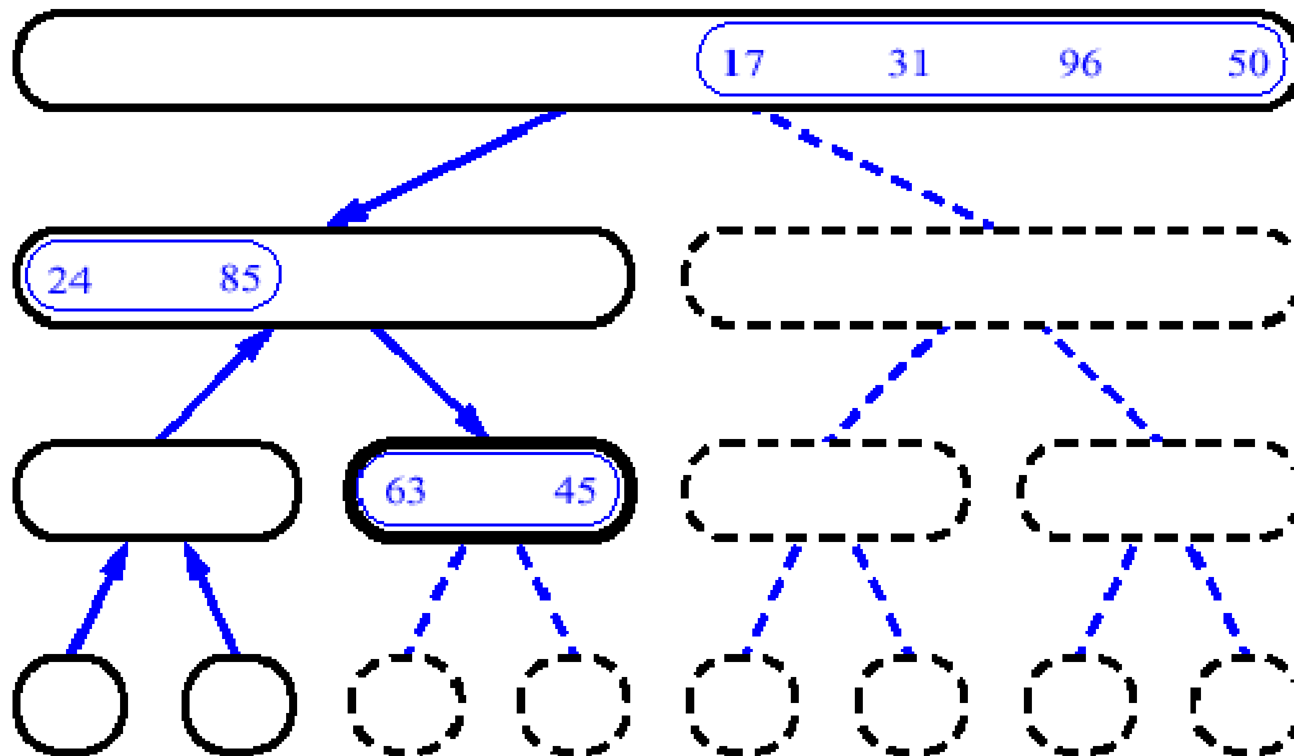
MergeSort (Example) - 9



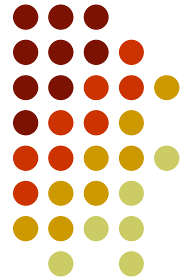


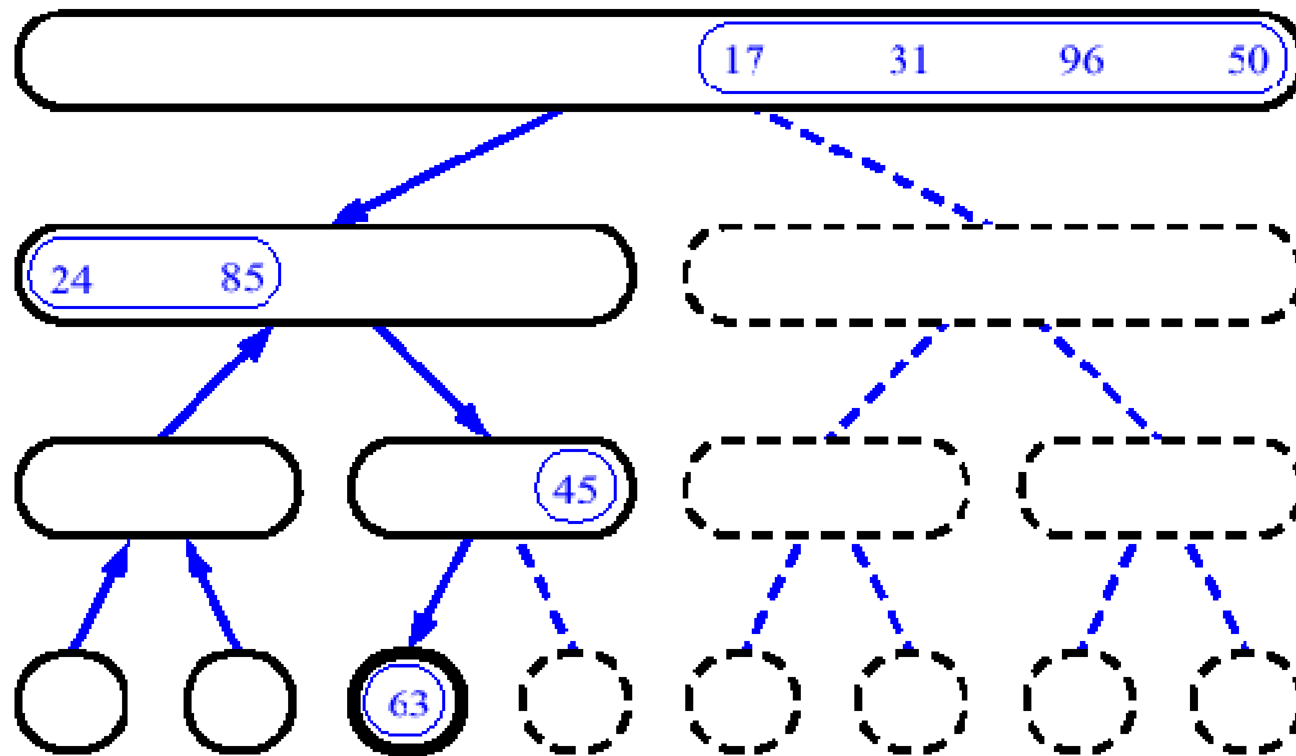
MergeSort (Example) - 10



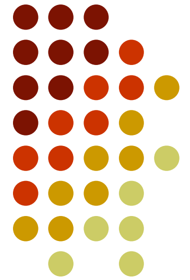


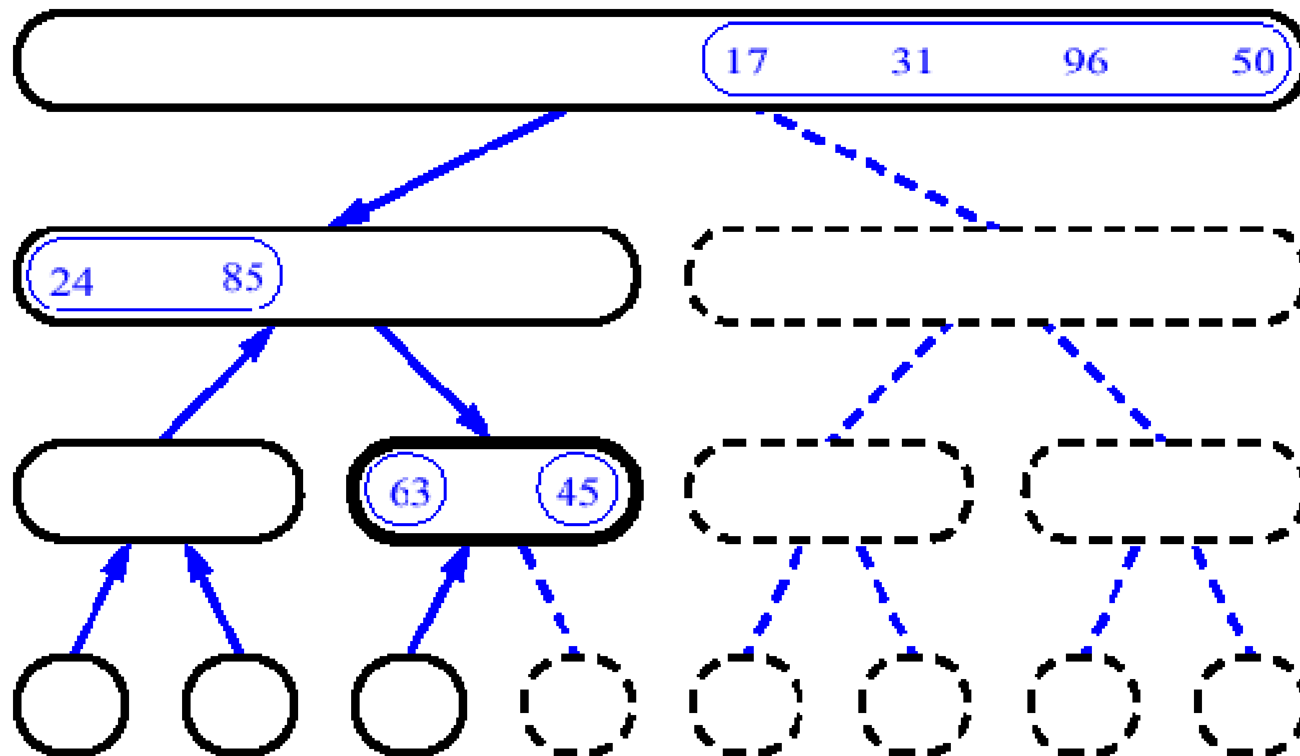
MergeSort (Example) - 11



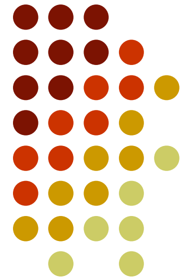


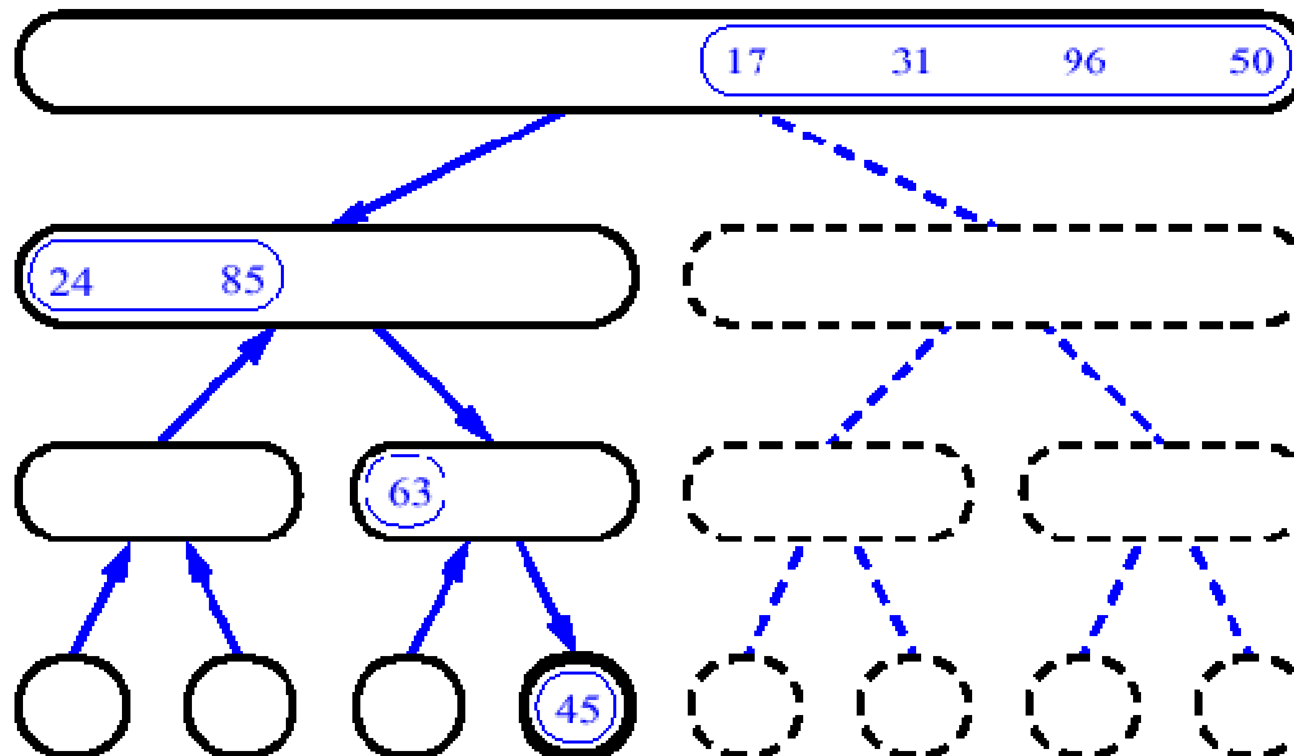
MergeSort (Example) - 12



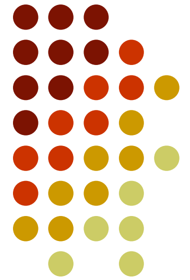


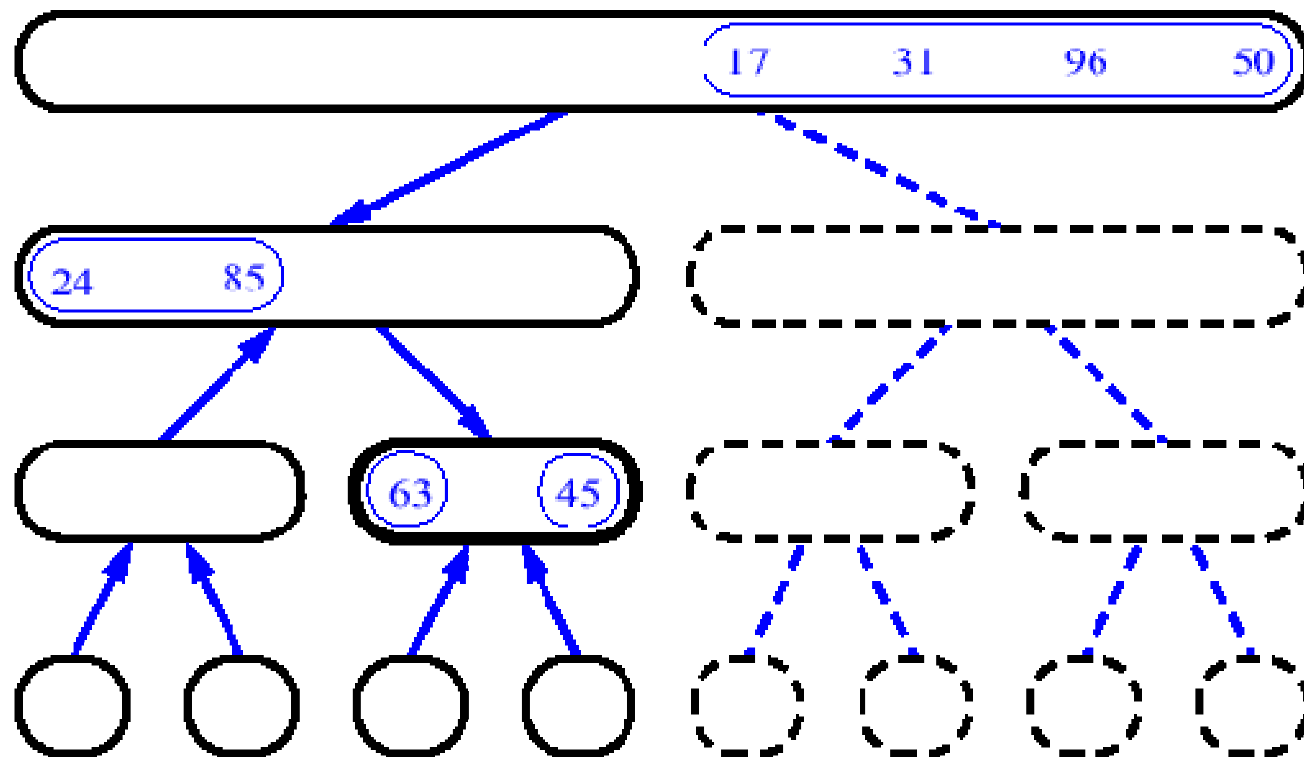
MergeSort (Example) - 13



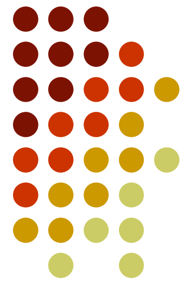


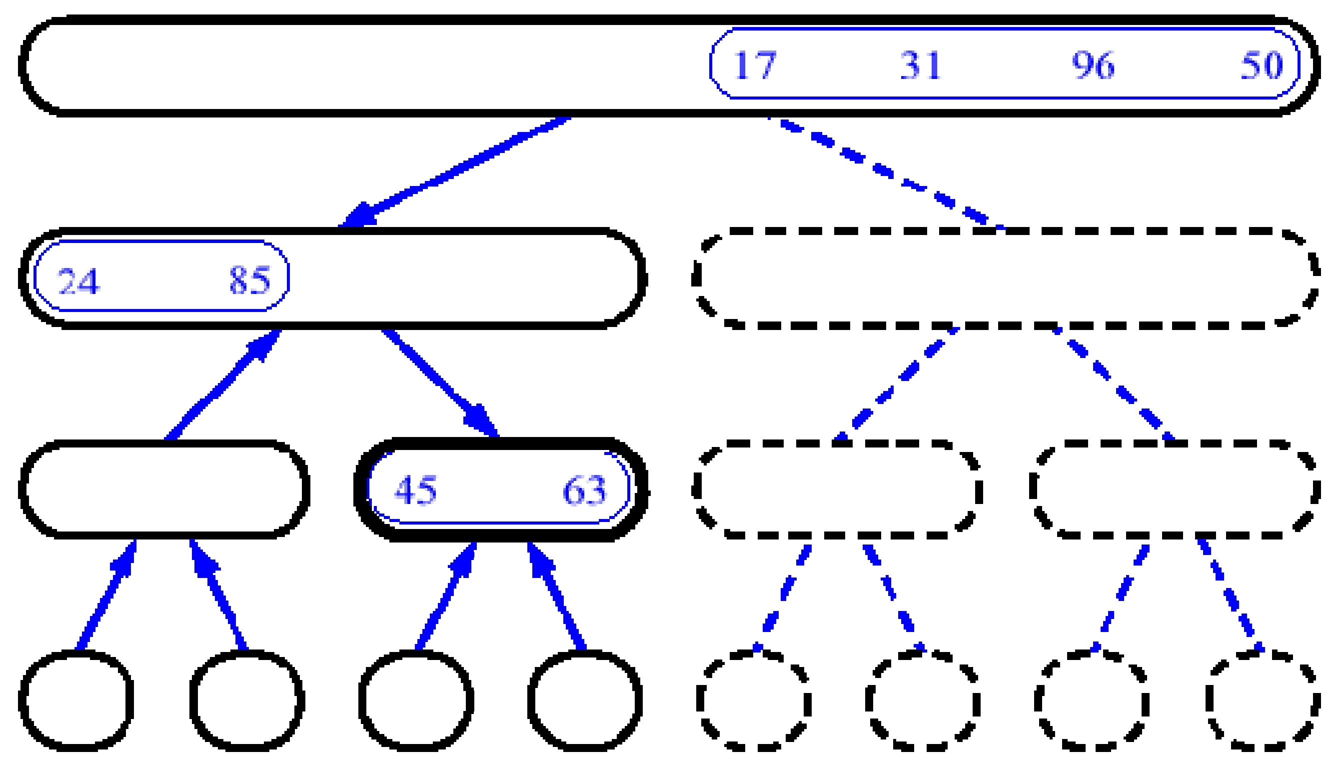
MergeSort (Example) - 14



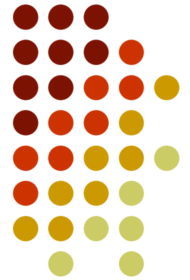


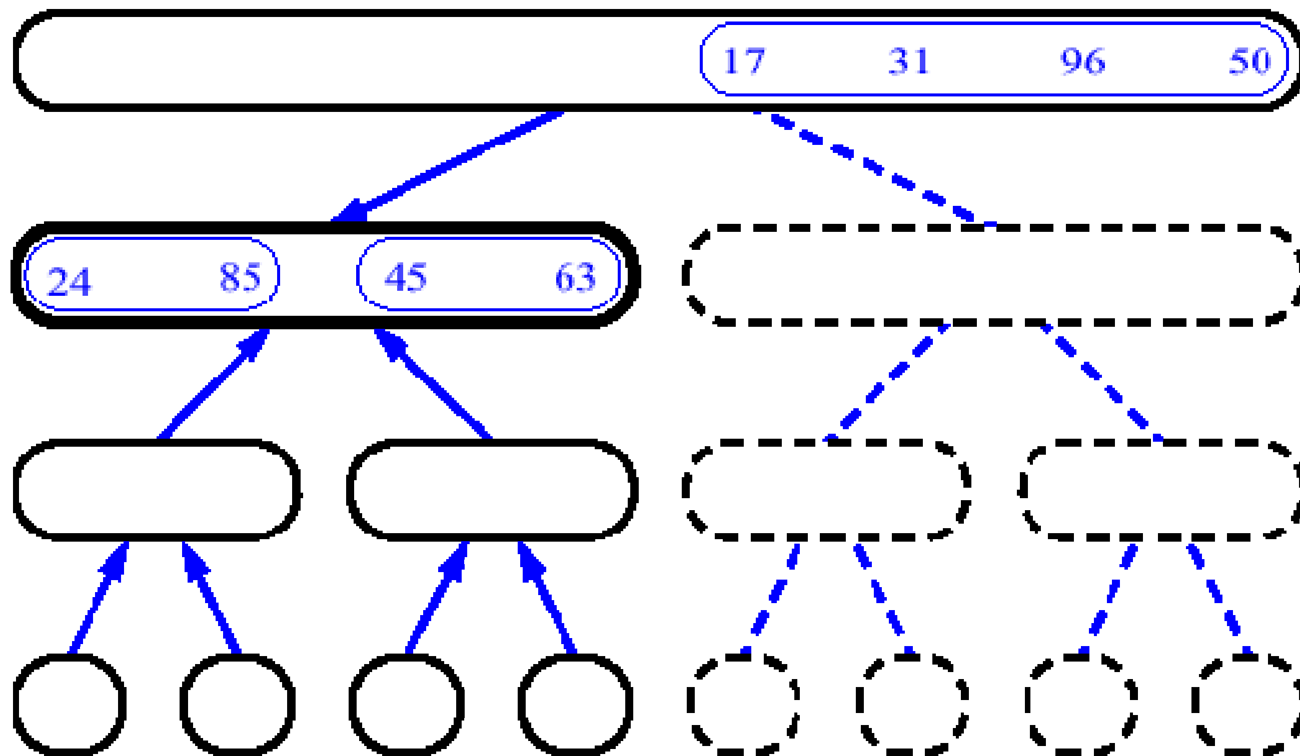
MergeSort (Example) - 15



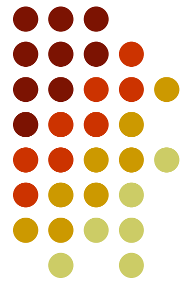


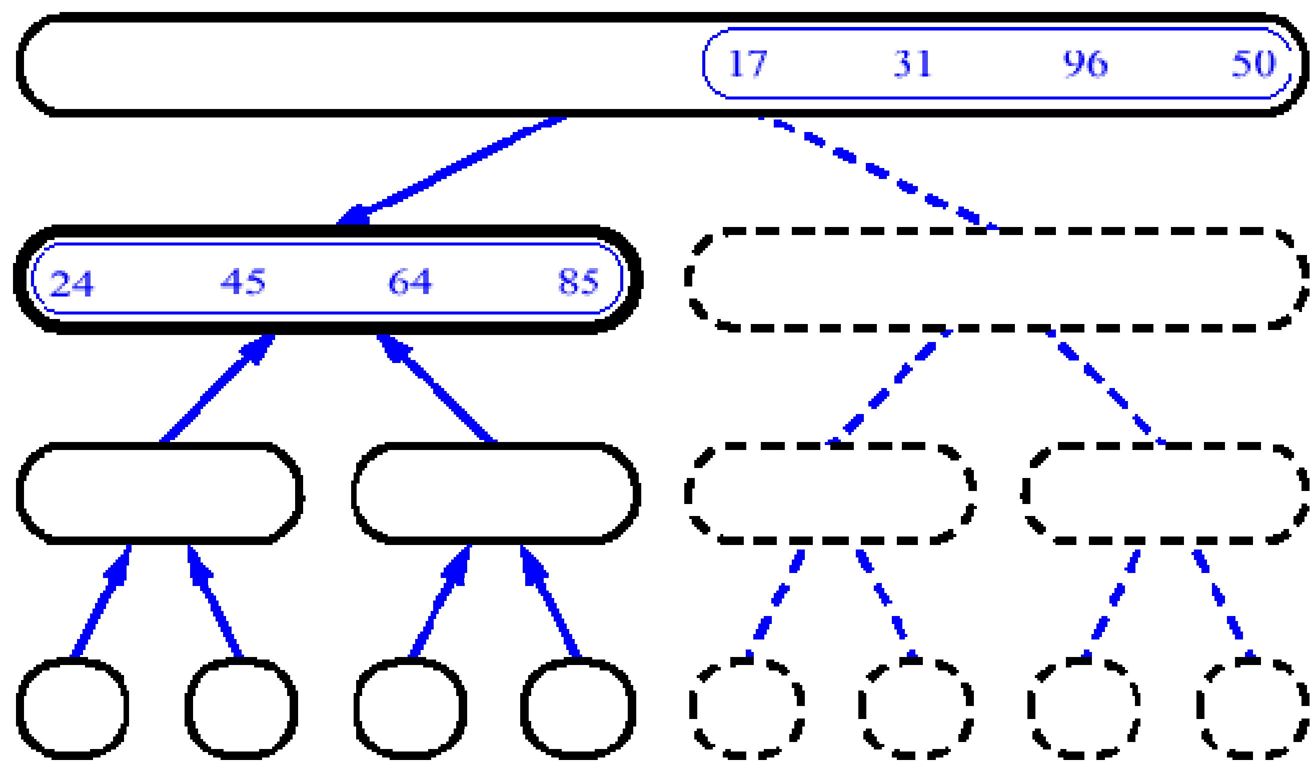
MergeSort (Example) - 16



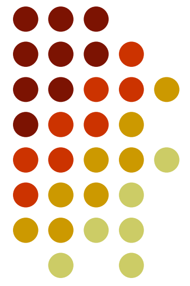


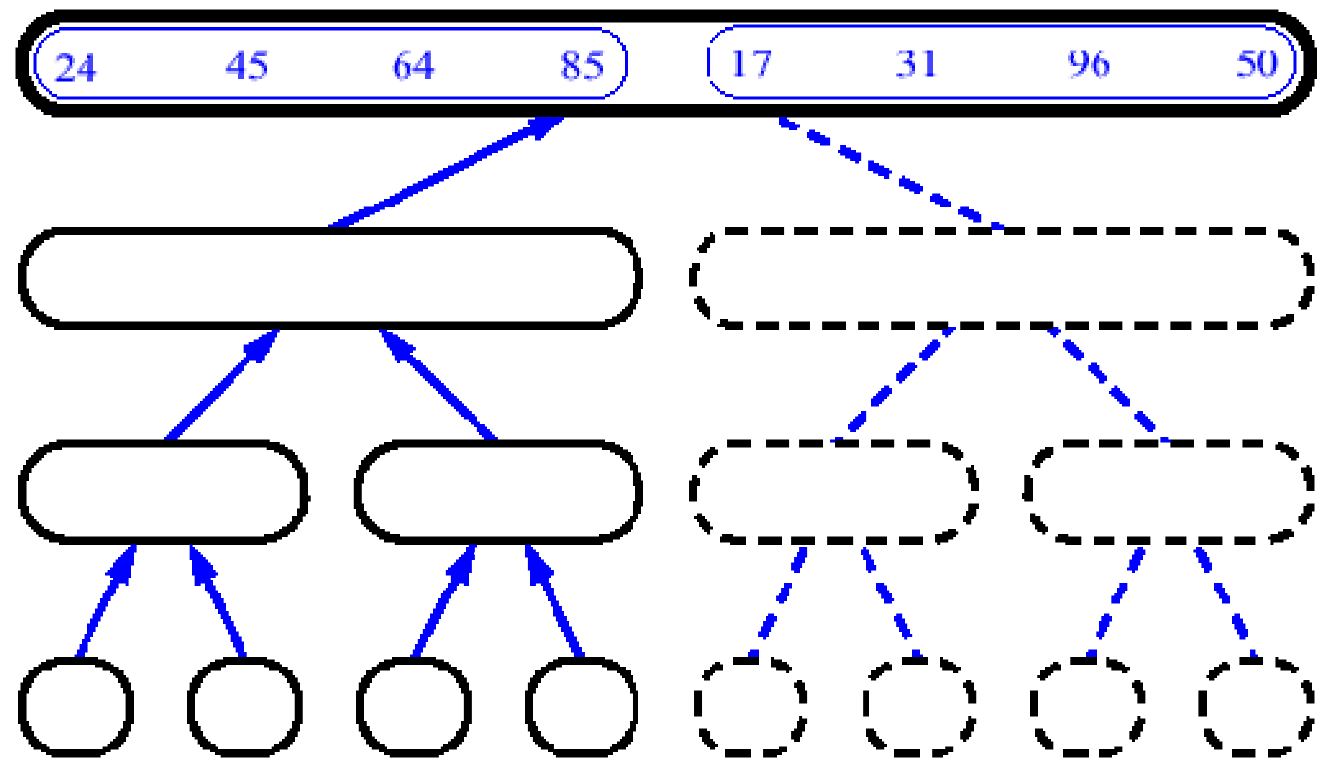
MergeSort (Example) - 17



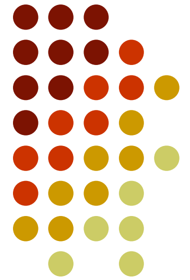


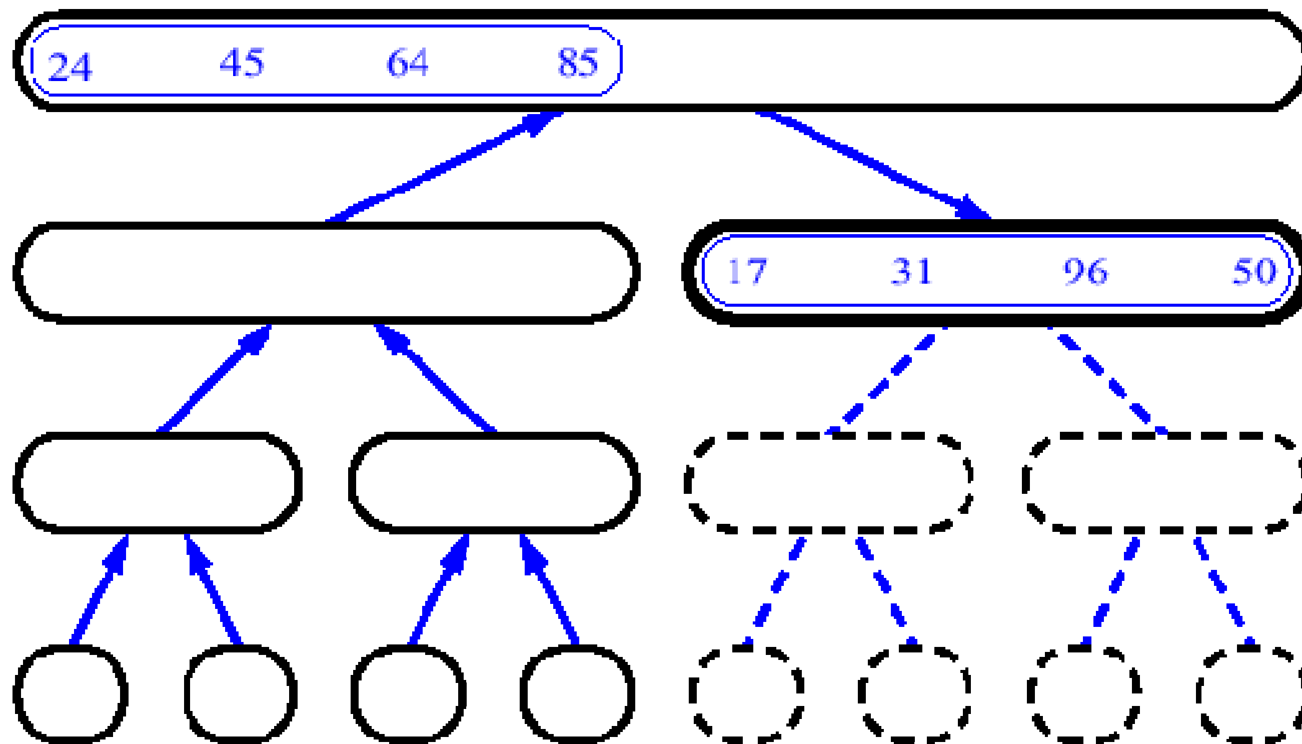
MergeSort (Example) - 18



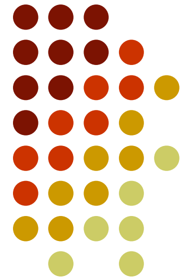


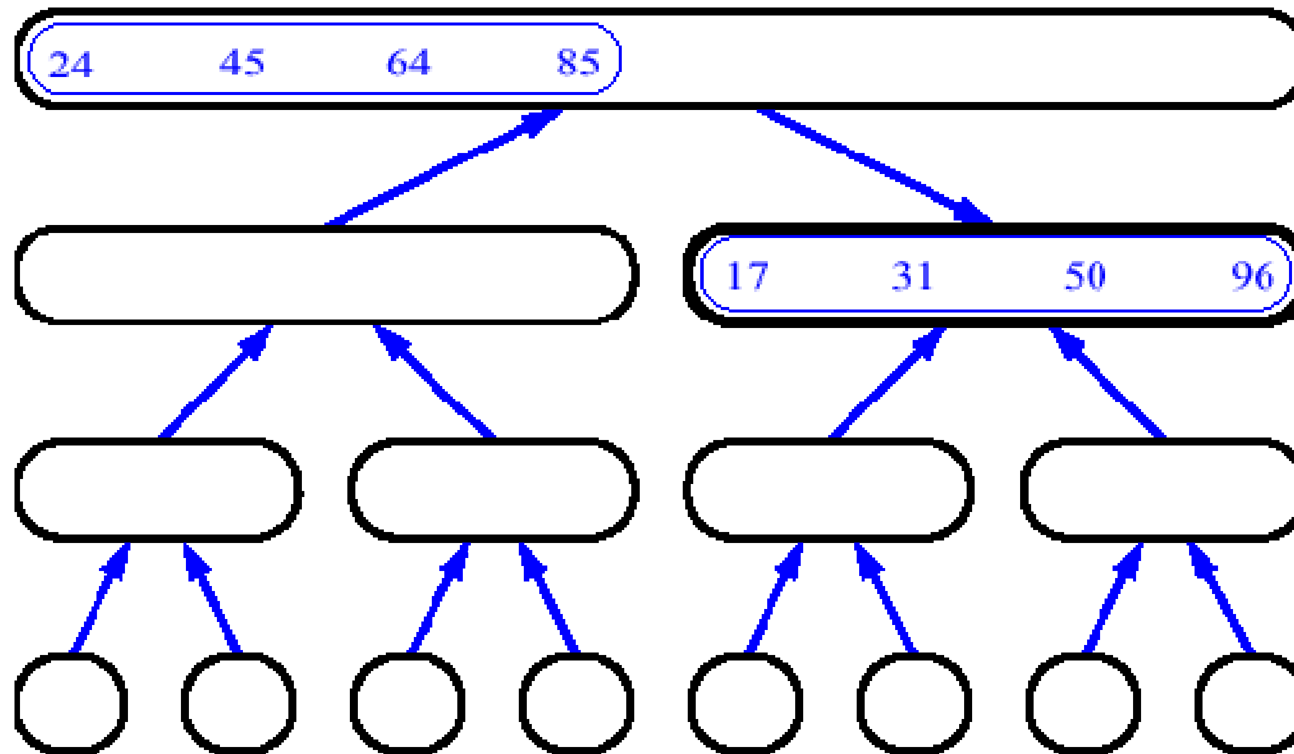
MergeSort (Example) - 19



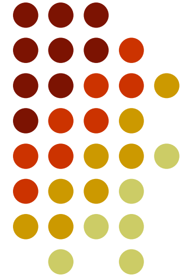


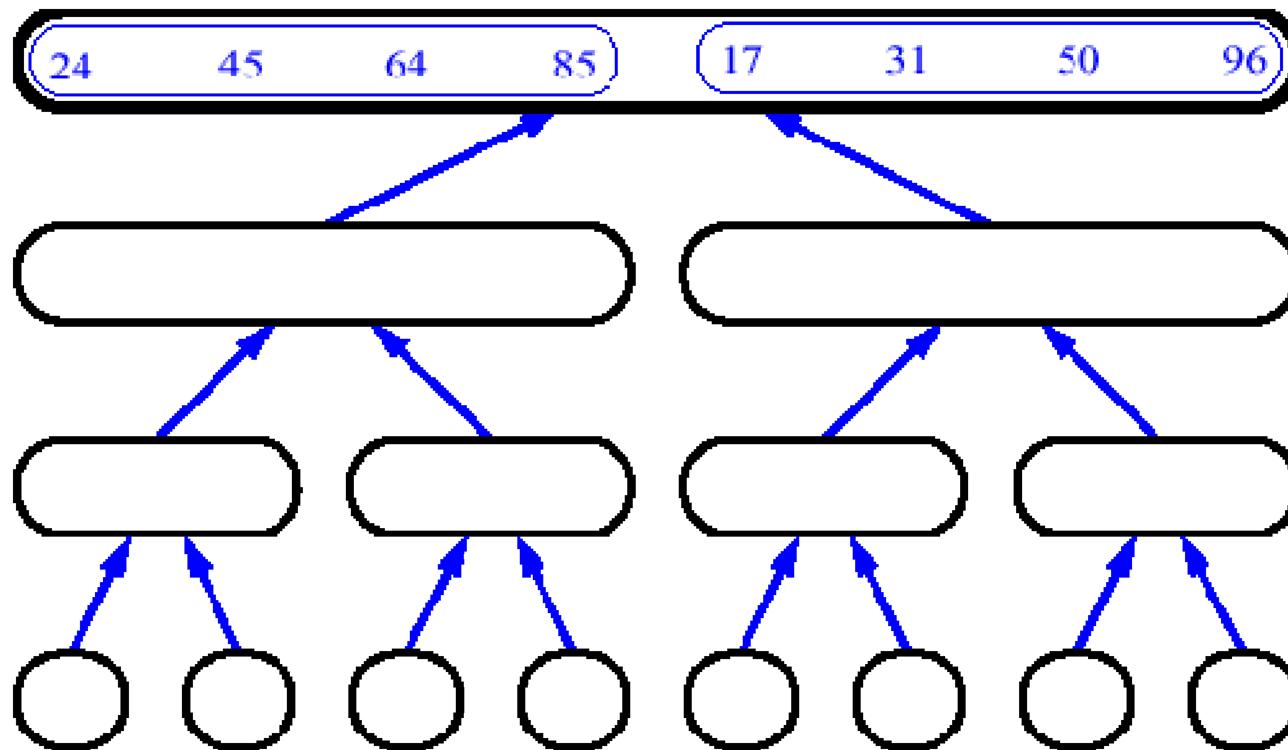
MergeSort (Example) - 20



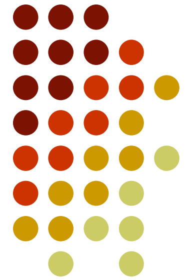


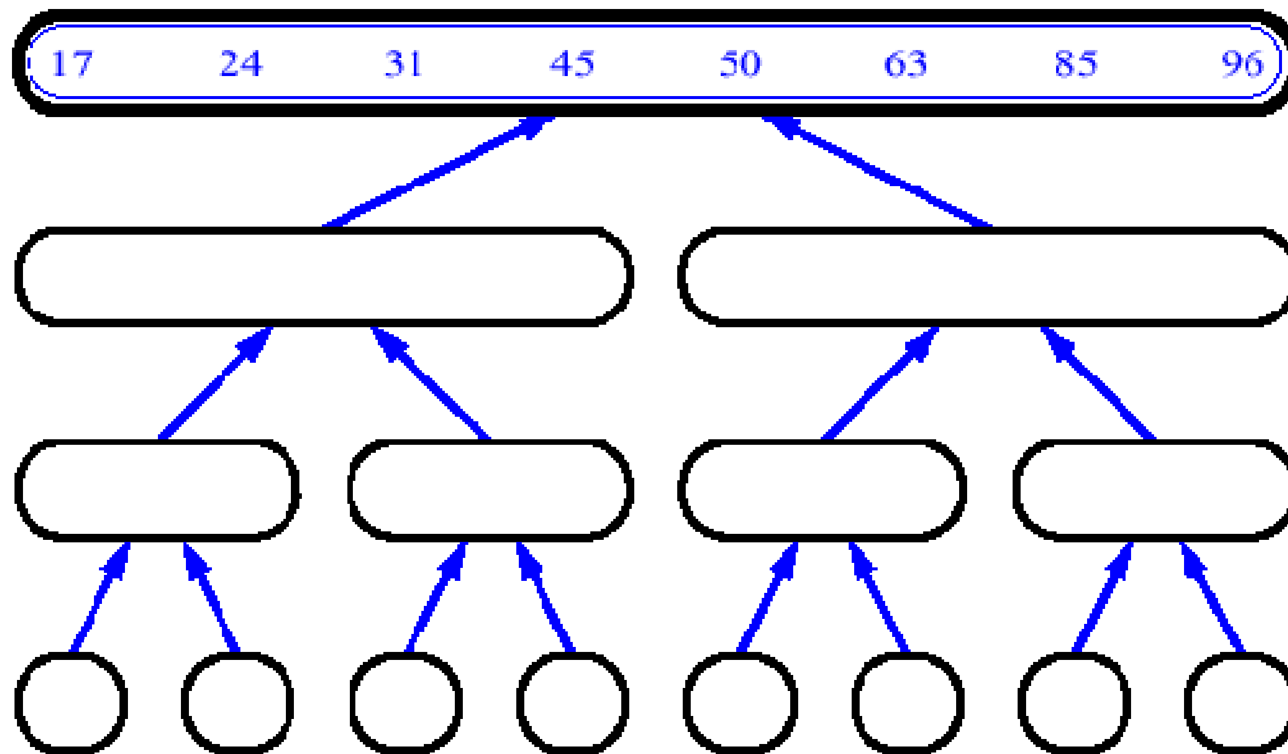
MergeSort (Example) - 21



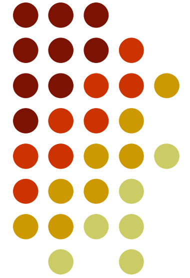


MergeSort (Example) - 22



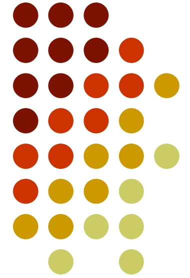


Merge Sort

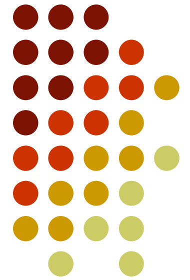


```
private void MergeSort(Comparable[] arr, int lowerBound,
                              int upperBound)
{
    if (lowerBound <= upperBound)       // if range is 0 or 1,
        return;                          // no need to sort
    else
    {
        // find midpoint
        int mid = (lowerBound+upperBound) / 2;
        // sort low half
        MergeSort(arr, lowerBound, mid);
        // sort high half
        MergeSort(arr, mid+1, upperBound);
        // merge them
        merge(arr, lowerBound, mid, upperBound);
    }  // end else
}  // end MergeSort()
```

Merge Sort: merge

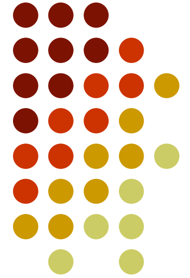


```
private void merge(Comparable[] arr, int low1, int high1, int high2)
   {
     int n = high2 - low1 + 1;              // # of items
     Comparable[] tmp=new Comparable[n];    // tmp array
     int j = 0;                             // tmp index
     int low2 = high1 + 1;
     int i1 = low1;                         // index in the first part
     int i2 = low2;
                                           // index in the secodn part

     while (i1 <= high1 && i2 <= high2)
        if (arr[i1].compareTo(arr[i2]) < 0)
           tmp[j++] = arr[i1++];
        else
           tmp[j++] = arr[i2++];

     while (i1 <= high1) // copy remaining elements in the first part
        tmp[j++] = arr[i1++];

     while (i2 <= high2) // copy remaining elements in the second part
        tmp[j++] = arr[i2++];

     for (j=0; j<n; j++)  // copy everything back to original array
        arr[low1+j] = tmp[j];
   }  // end merge()
```



14	23	45	98

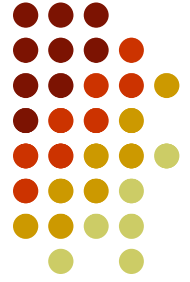
6	33	42	67

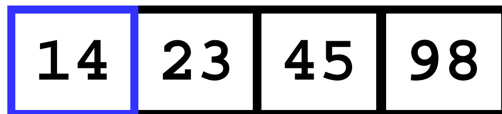


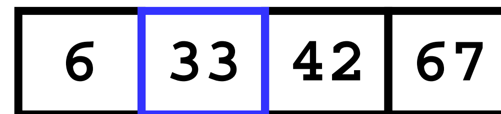
14	23	45	98

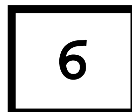
6	33	42	67

Merge

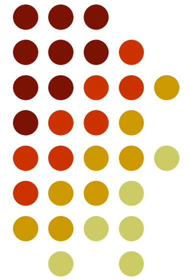








Merge

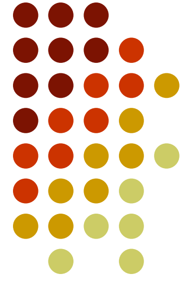


14	23	45	98

6	33	42	67

6	14

Merge

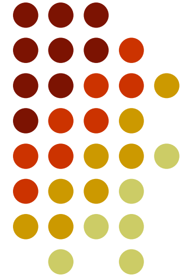


14	23	45	98

6	33	42	67

6	14	23

Merge

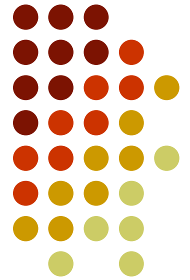


14	23	45	98

6	33	42	67

6	14	23	33

Merge

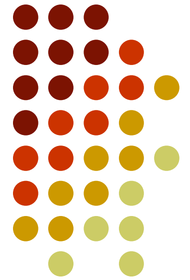


14	23	45	98

6	33	42	67

6	14	23	33	42

Merge

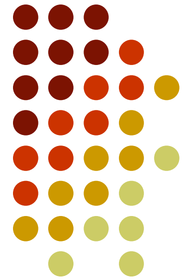


14	23	45	98

6	33	42	67

6	14	23	33	42	45

Merge

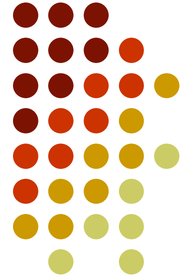


14	23	45	98

6	33	42	67

6	14	23	33	42	45	67

Merge

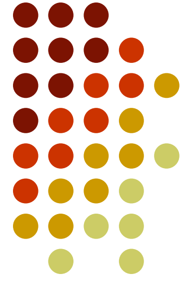


14	23	45	98

6	33	42	67

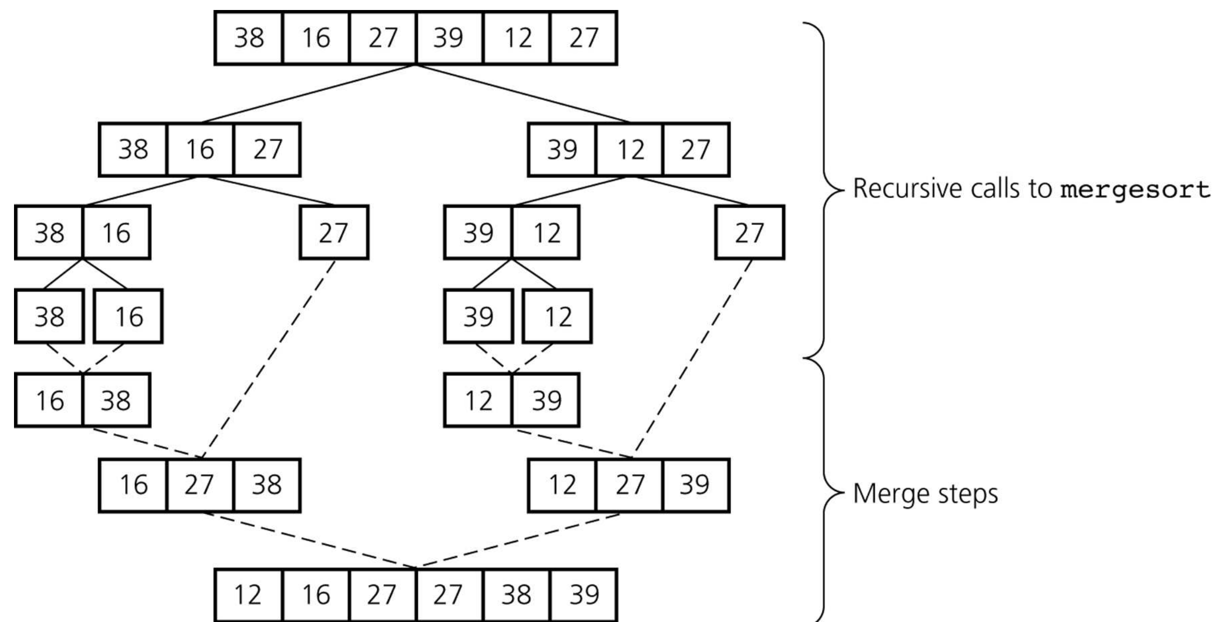
6	14	23	33	42	45	67	98

Merge

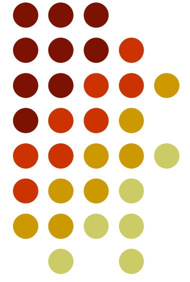


Merge Sort Summarized

- To sort n numbers
  - if n=1 done!
  - recursively sort 2 lists of numbers $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ elements
  - merge 2 sorted lists in $O(n)$ time



Running time of MergeSort

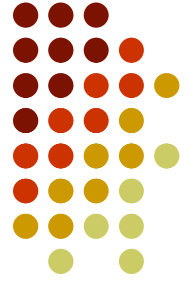


- The running time can be expressed as a recurrence:

$$T(n) = \begin{cases} \text{solving_trivial_problem} & \text{if } n = 1 \\ \text{num_pieces } T(n/\text{subproblem_size_factor}) + \text{dividing} + \text{combining} & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Repeated Substitution Method



$T(n) = 2T(n/2) + cn \qquad n > 1$

$\quad\;\; = 1 \qquad\qquad\qquad\quad n=1$

- $T(n) = 2T(n/2) + cn$

  $= 2\ \{ 2T(n/2^2) + c.n/2\} + cn$

  $= 2^2\ T(n/2^2) + c.2n$

  $= 2^2\ \{2T(n/2^3) + c.n/2^2\} + c.2n$

  $= 2^3\ T(n/2^3) + c.3n$

  $= \ldots\ldots$

  $= 2^k\ T(n/2^k) + c.k \cdot n$

  $= \ldots.$

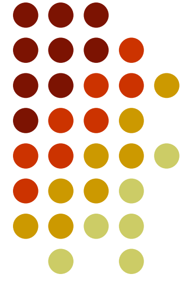
  $= 2^{\log n}\ T(1) + c.(\log n) \cdot n \quad \text{when } n/2^k = 1 \Rightarrow k = \log_2 n$

  $= 2^{\log n} \cdot 1 + c.(\log n) \cdot n$

  $= n + c.n \log n \qquad \text{where } 2^{\log n} = n$

- Therefore, $T(n) = O(n \log n)$

The Substitution method



$\boldsymbol{T(n) = 2T(n/2) + cn}$

- **Guess**: $T(n) = O(n \log n)$
- Proof:

Prove that $T(n) \leq d\, n \log n$ for $d>0$

$T(n) \leq 2(d \cdot n/2 \cdot \log n/2) + cn$

where $T(n/2) \leq d \cdot n/2\ (\log n/2)$ for $d>0$

$\leq dn \log n/2 + cn$

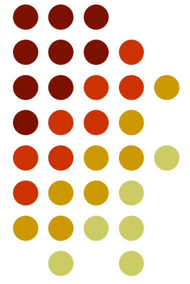
$= dn \log n - dn + cn$

$= dn \log n + (c-d)n$

$\leq dn \log n$ if $d \geq c$

- Therefore, $\boldsymbol{T(n) = O(n \log n)}$

The Master Method (optional)



- Master Theorem
  - Let $a \geq 1$, $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

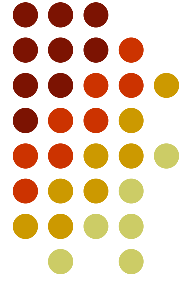
    $$T(n) = aT(n/b) + f(n),$$

    where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then T(n) can be bounded asymptotically as follows.

    1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

    2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$

    3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c<1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

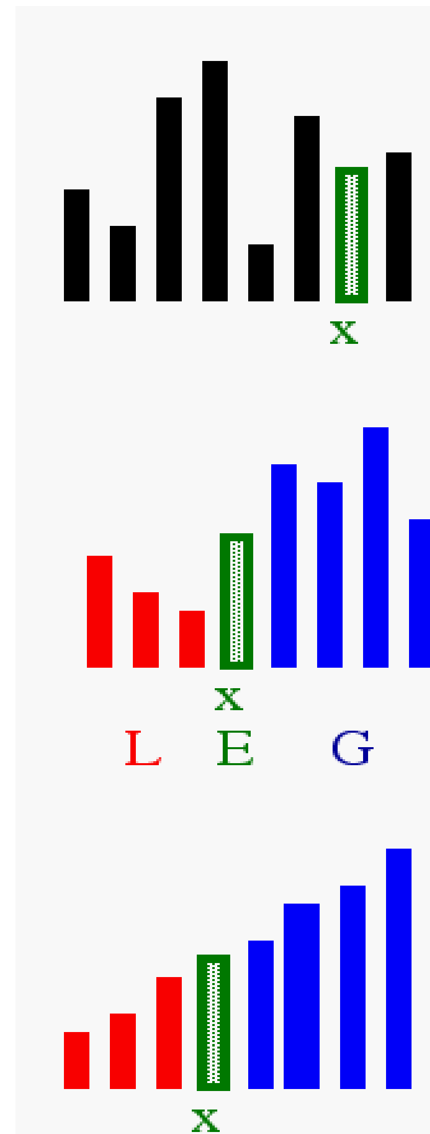


Quick Sort: Idea

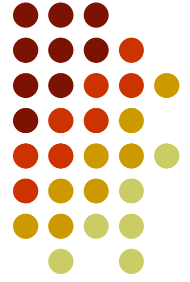
1) **Select**: pick an element

2**) Divide**: partition elements so that x goes to its final position E

3) **Conquer**: recursively sort left and right partitions

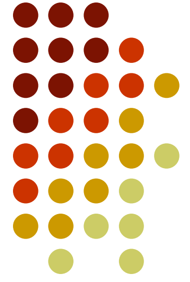


Quick Sort - algorithm

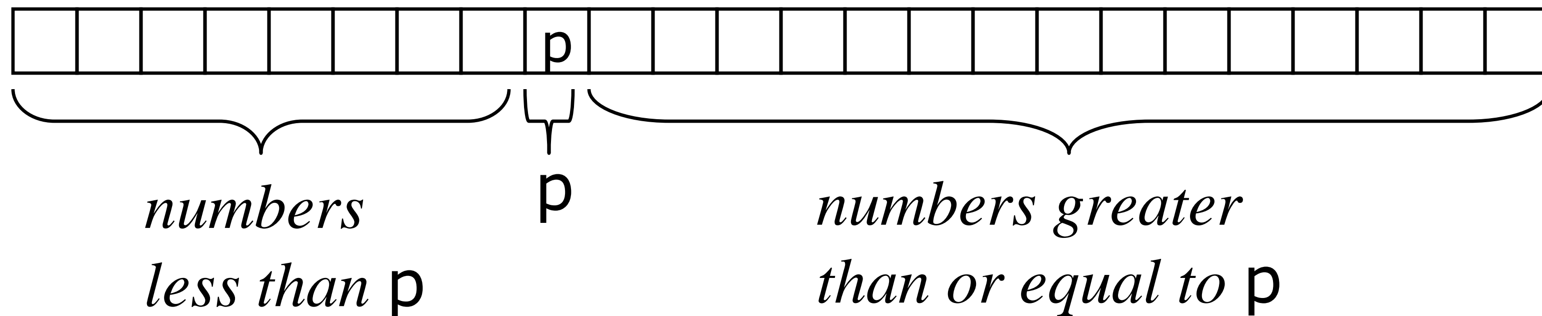


```
public void QuickSort(Comparable[] arr, int low, int
   high) {
     if (low <= high) // if size <= 1 already sorted
       return;
     else // size is 2 or larger
     {
       // partition range
       int pivotIndex = partition(arr, low, high);
       // sort left subarray
       QuickSort(arr, low, pivotIndex - 1);
       // sort right subarray
       QuickSort(arr, pivotIndex + 1, high);
     }
   }
```

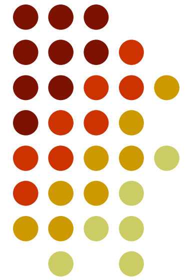
Quick Sort - Partitioning

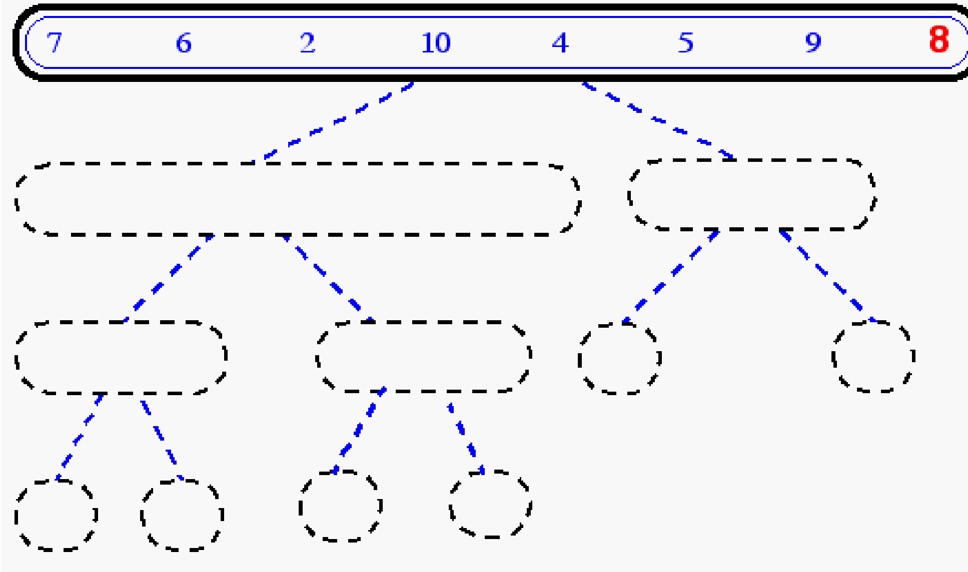


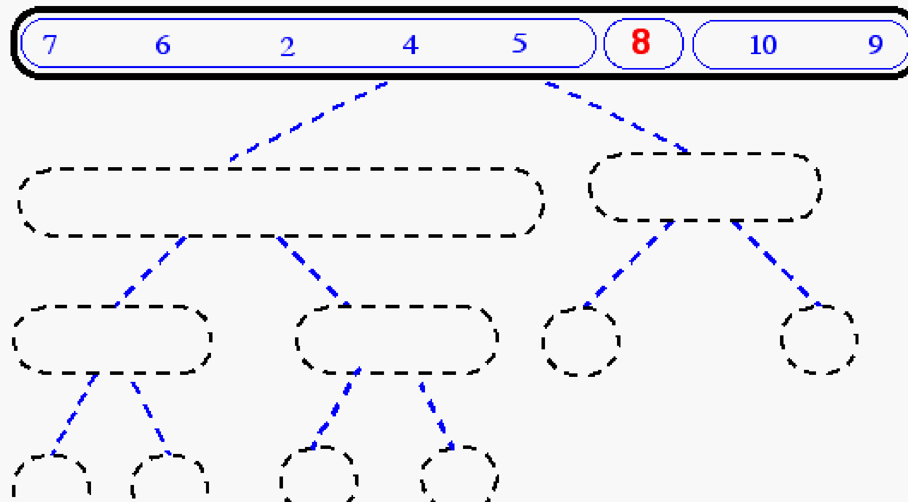
- A key step in the Quick sort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



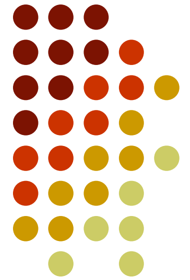
Quick Sort – Partitioning



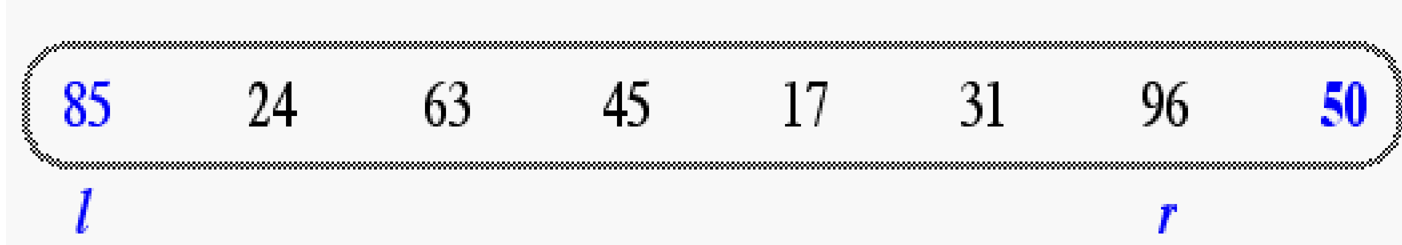




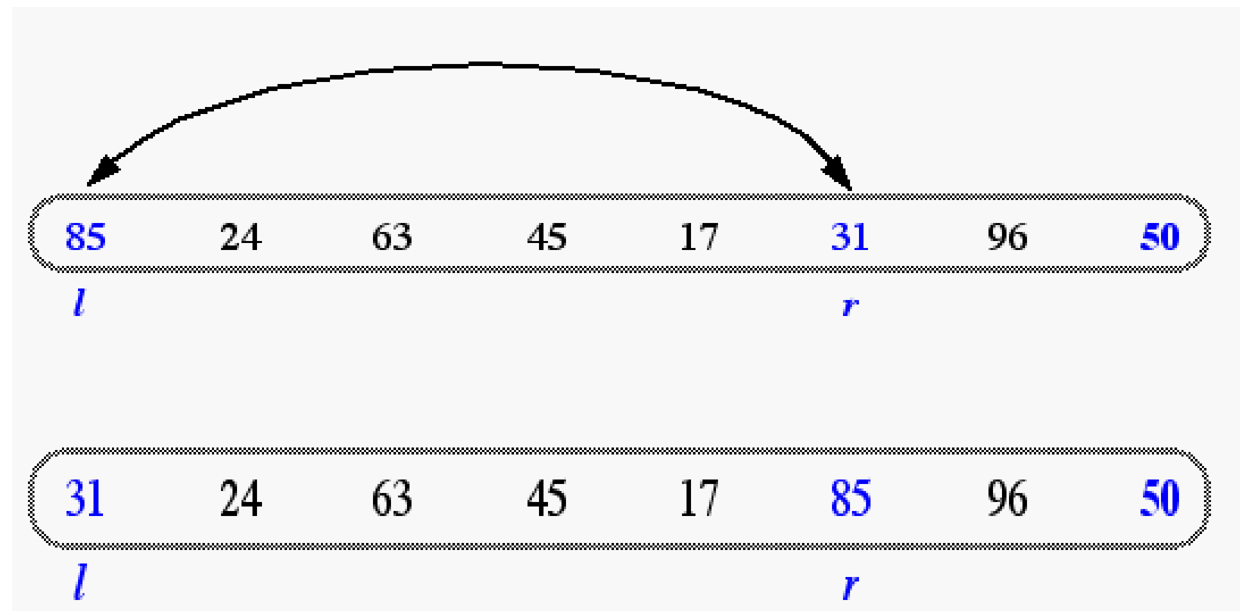
Quick Sort – Partitioning – algorithm



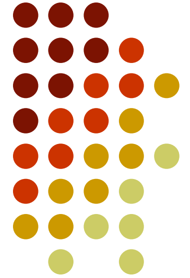
Index *l* scans the sequence from the left, and index *r* from the right.

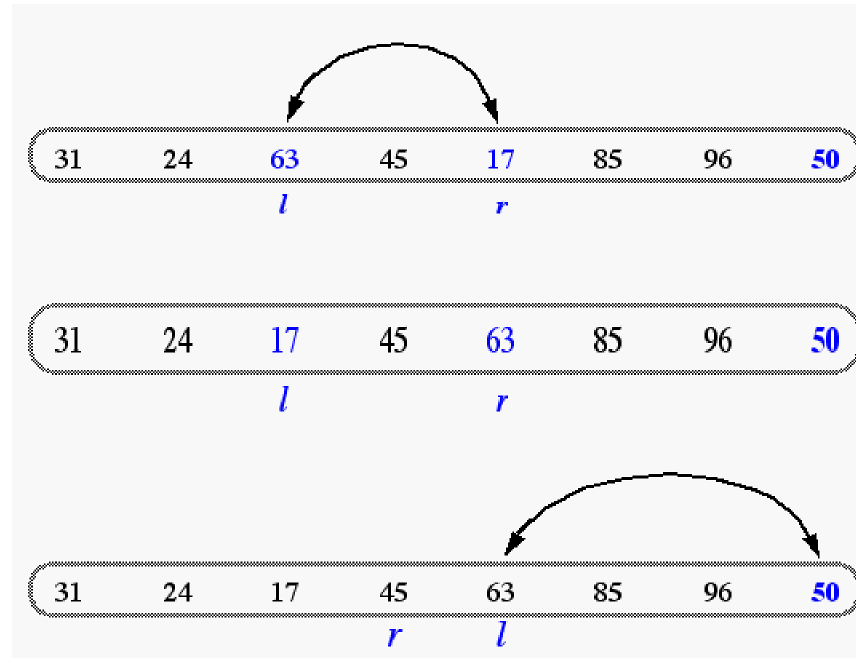


Increment *l* until *arr[l]* is larger than the pivot; and decrement *r* until *arr[r]* is smaller than the pivot.
Then swap elements indexed by *l* and *r*.
Repeat until whole array is processed.



Quick Sort – Partitioning – algorithm





A final swap with the pivot completes the partitioning.