Sorting

Maunendra Sankar Desarkar
IIT Hyderabad

Acknowledgements

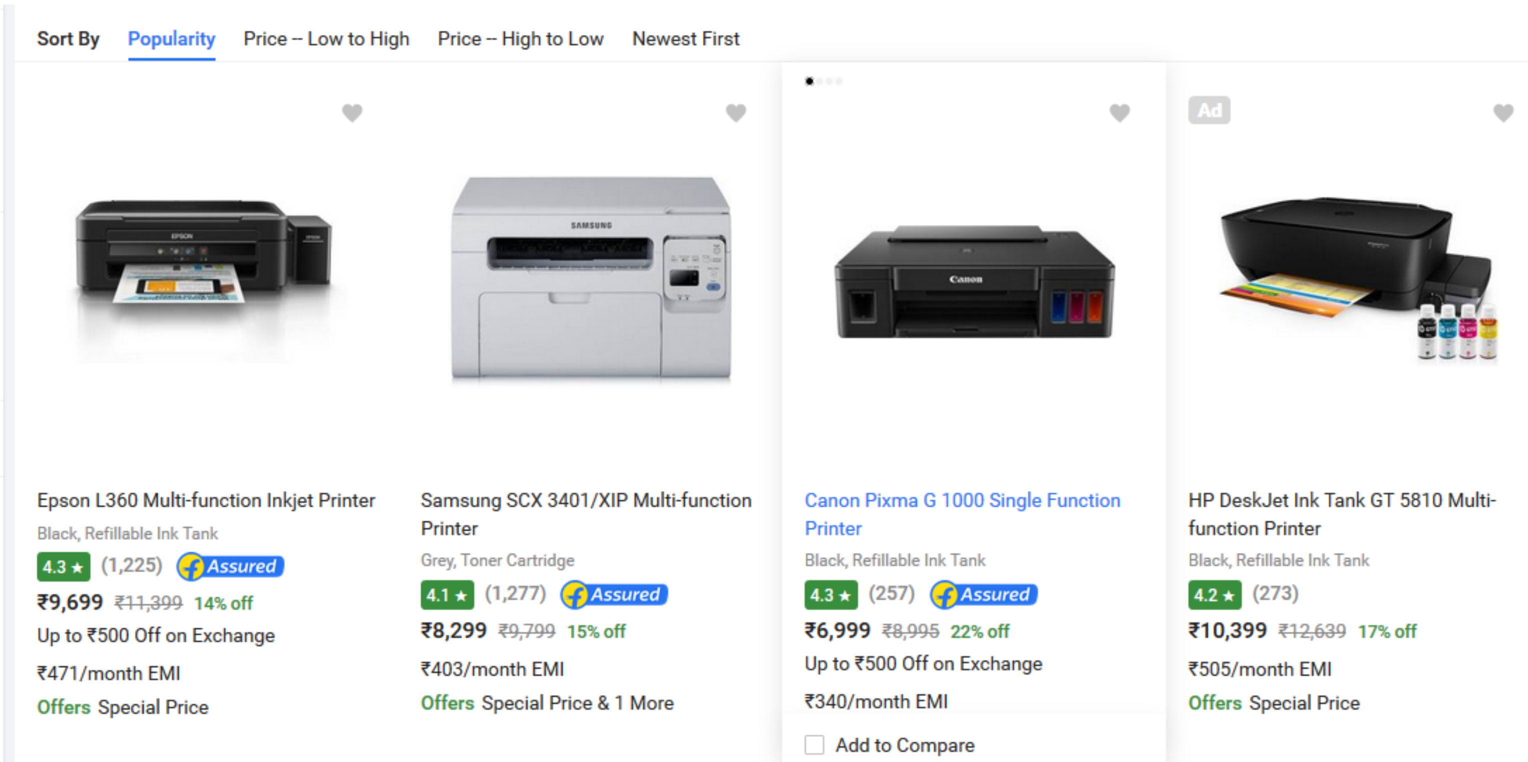
- Slides adapted from publicly available lectures on same topic by different instructors and researchers.

- Sorting: “arrange systematically in groups; separate according to type”
  - Categorizing/dividing
  - Arranging in specific order

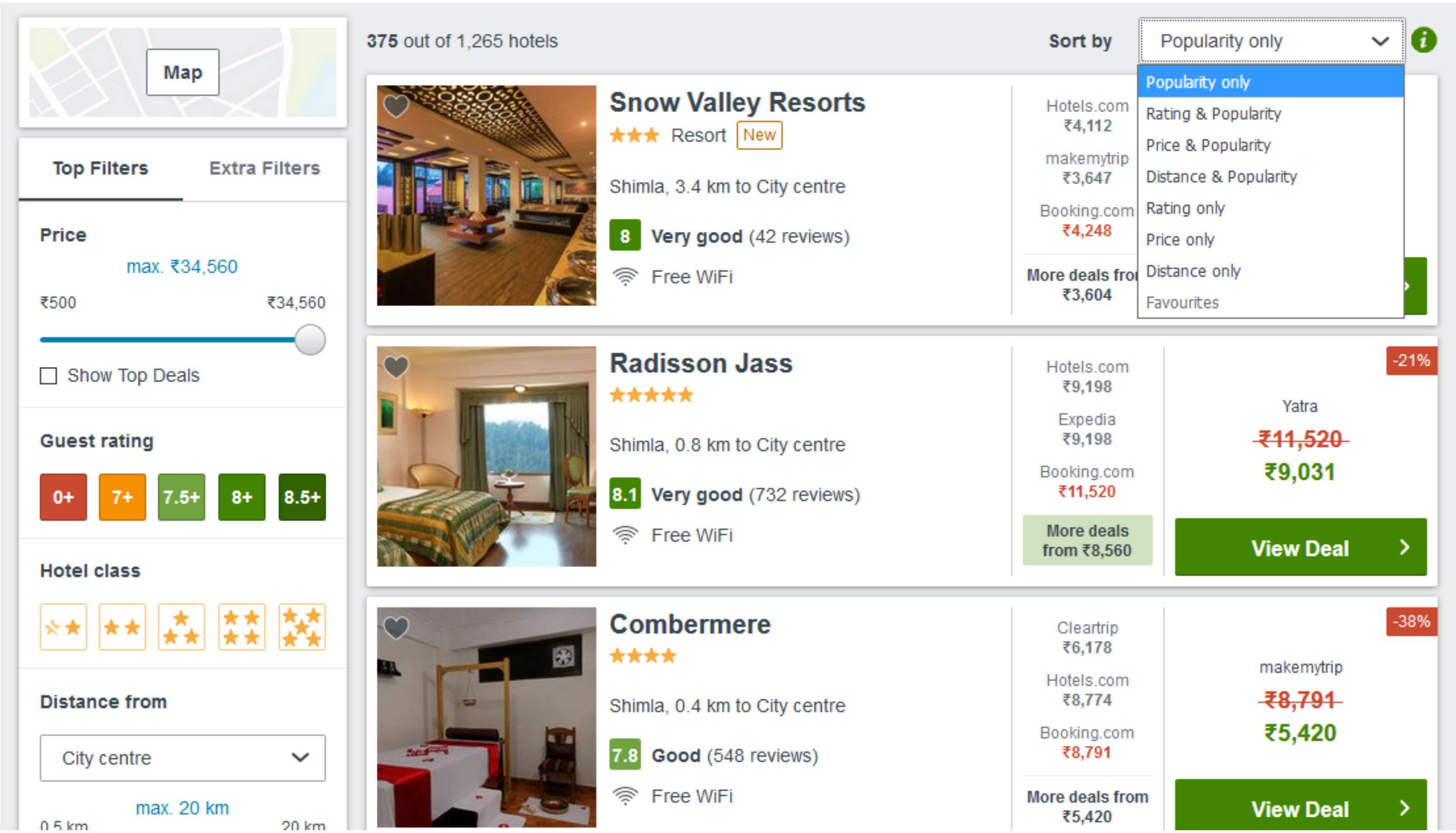
Sorting in Practice: Example 1

Train Number	Train Name	Origin	Departure Time	Destination	Arrival Time	Travel Time	Days of Run							Classes
							M	T	W	T	F	S	S	
12723	TELANGANA EXP	SC	06:50	NDLS	09:05	26:15	Y	Y	Y	Y	Y	Y	Y	1A 2A 3A SL
22691	RAJDHANI EXP	SC	07:50	NZM	05:55	22:05	Y	Y	Y	Y	Y	Y	Y	1A 2A 3A
12285	NZM DURONTO EX	SC	13:10	NZM	10:35	21:25	N	N	N	Y	N	N	Y	1A 2A 3A SL
12721	NIZAMUDDIN EXP	SC	23:00	NZM	04:05	29:05	Y	Y	Y	Y	Y	Y	Y	2A 3A SL
12649	SAMPARK KRANTI	KCG	08:30	NZM	09:15	24:45	Y	Y	N	Y	N	Y	Y	1A 2A 3A SL

Sorting in Practice: Example 2



Sorting in Practice: Example 3



Why Sorting?

- Practical applications (few more)
  - People by last name
  - Search engine results by predicted relevance
  - Sort customer reviews based on date/helpfulness/…
  - Folder viewer, Task manager ..
  - …

- Fundamental to other algorithms
- Data pre-processing
- If we do the work now, future operations may be faster

- Different algorithms have different asymptotic and constant-factor trade-offs
  - No single 'best' sort for all scenarios
  - Knowing one way to sort just isn't enough

Problem statement

- There are *n* comparable elements in an array and we want to rearrange them to be in increasing order

- Pre:
  - An array **`A`** of data records
  - A value in each data record
  - A comparison function
    - `<`, `=`, `>`, `compareTo`

- Post:
  - For each distinct position **`i`** and **`j`** of **`A`**, if **`i<j`** then **`A[i]`** $\leq$ **`A[j]`**
  - **`A`** has all the same data it started with

A crude way of sorting

- Order a list of values by repetitively shuffling them and checking if they are sorted

- more specifically:
  - scan the list, seeing if it is sorted
  - if not, shuffle the values in the list and repeat

- Performance?

More Definitions

In-Place Sort:

A sorting algorithm is in-place if it requires only O(1) extra space to sort the array.

- Usually modifies input array
- Can be useful: lets us minimize memory

Stable Sort:

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort.

- Items that 'compare' the same might not be exact duplicates
- Might want to sort on some, but not all attributes of an item
- Can be useful to sort on one attribute first, then another one

Stable Sort Example

Input:

```
[(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
```

Compare function: compare pairs by number only

Output (stable sort):

```
[(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]
```

Output (unstable sort):

```
[(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]
```

Bubble sort

- **bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

- more specifically:
  - scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order

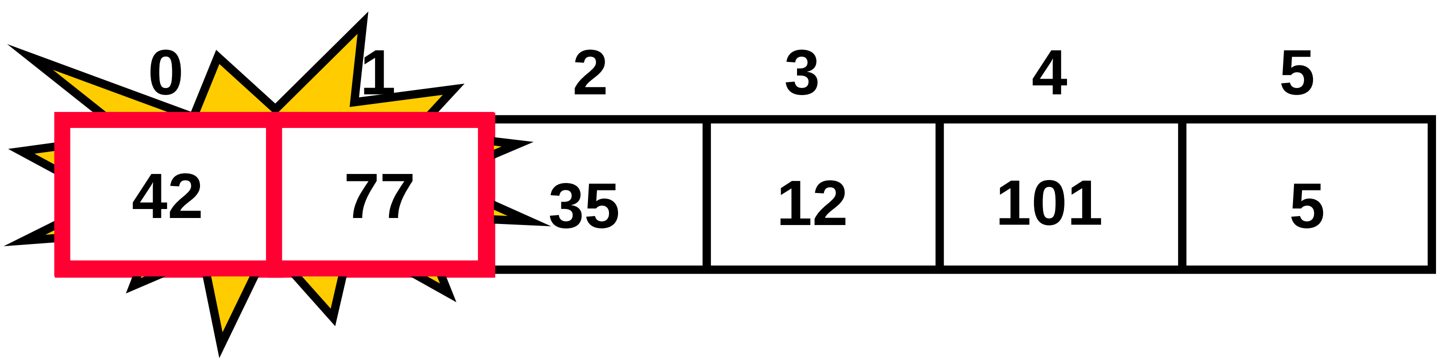
Bubble sort

- **bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

- more specifically:
  - scan the list, exchanging adjacent elements if they are not in relative order (are inversions); this bubbles the highest value to the end
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order

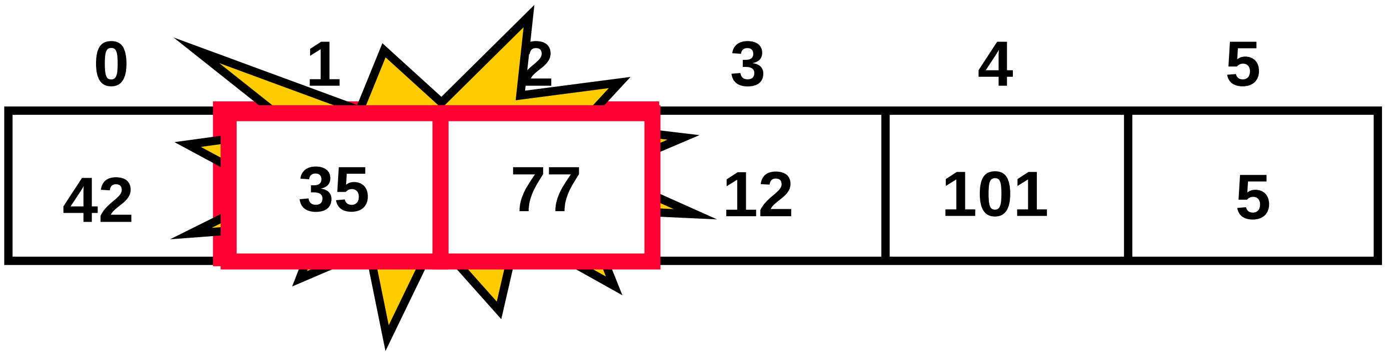
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



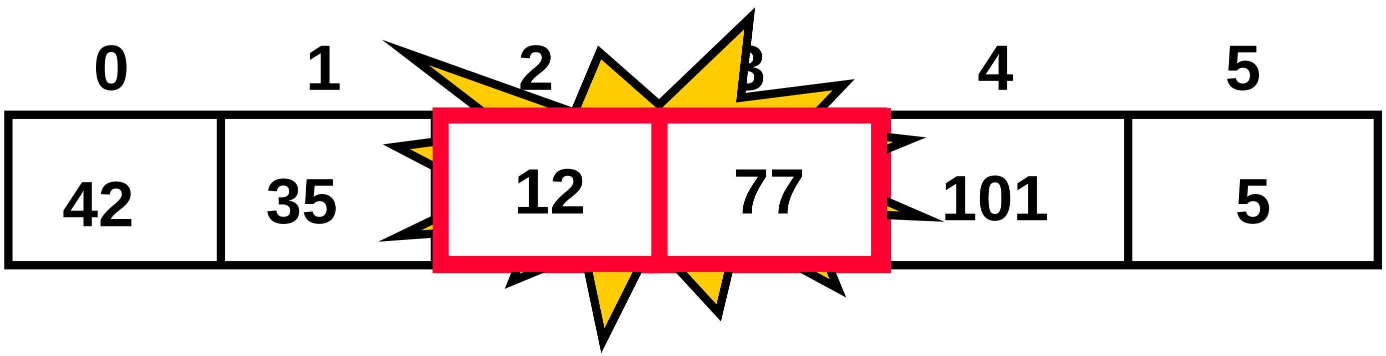
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling" largest element

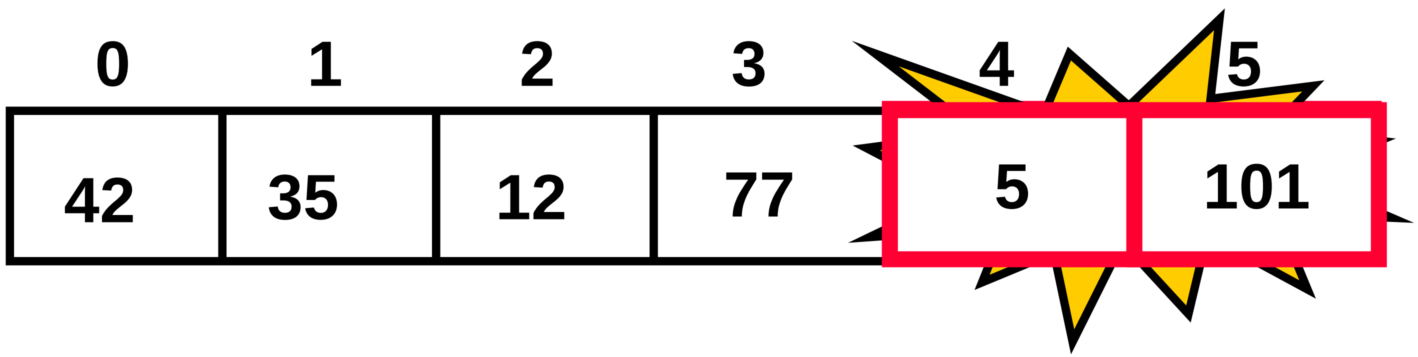
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

0	1	2	3	4	5
42	35	12	77	101	5

No need to swap

"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

0	1	2	3	4	5
42	35	12	77	5	$\boxed{101}$

Largest value correctly placed

Next steps?

Bubble sort code

```
void BubbleSort(int arr[], int n)
{
    int i, j, t;
    for (i = 0; ______________________; i++)

        for (j = 0; _____________; j++)
            if (__________________) {

                // swap arr[j] and arr[j+1]
                _________________________;
                _________________________;
                _________________________;
            }

}
```

Bubble sort code

```
void BubbleSort(int arr[], int n)
{
   int i, j, t;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in appropriate place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1]) {

              // swap arr[j] and arr[j+1]
              t = arr[j];
              arr[j]=arr[j+1];
              arr[j+1]=t;
           }
}
```

Bubble sort runtime

- Running time (# comparisons) for input size *n*:

$$
\begin{aligned}
\sum_{i=0}^{n-1}\sum_{j=1}^{n-1-i} 1 &= \sum_{i=0}^{n-1}(n-1-i) \\
&= n\sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i \\
&= n^2 - n - \frac{(n-1)n}{2} \\
&= \Theta(n^2)
\end{aligned}
$$

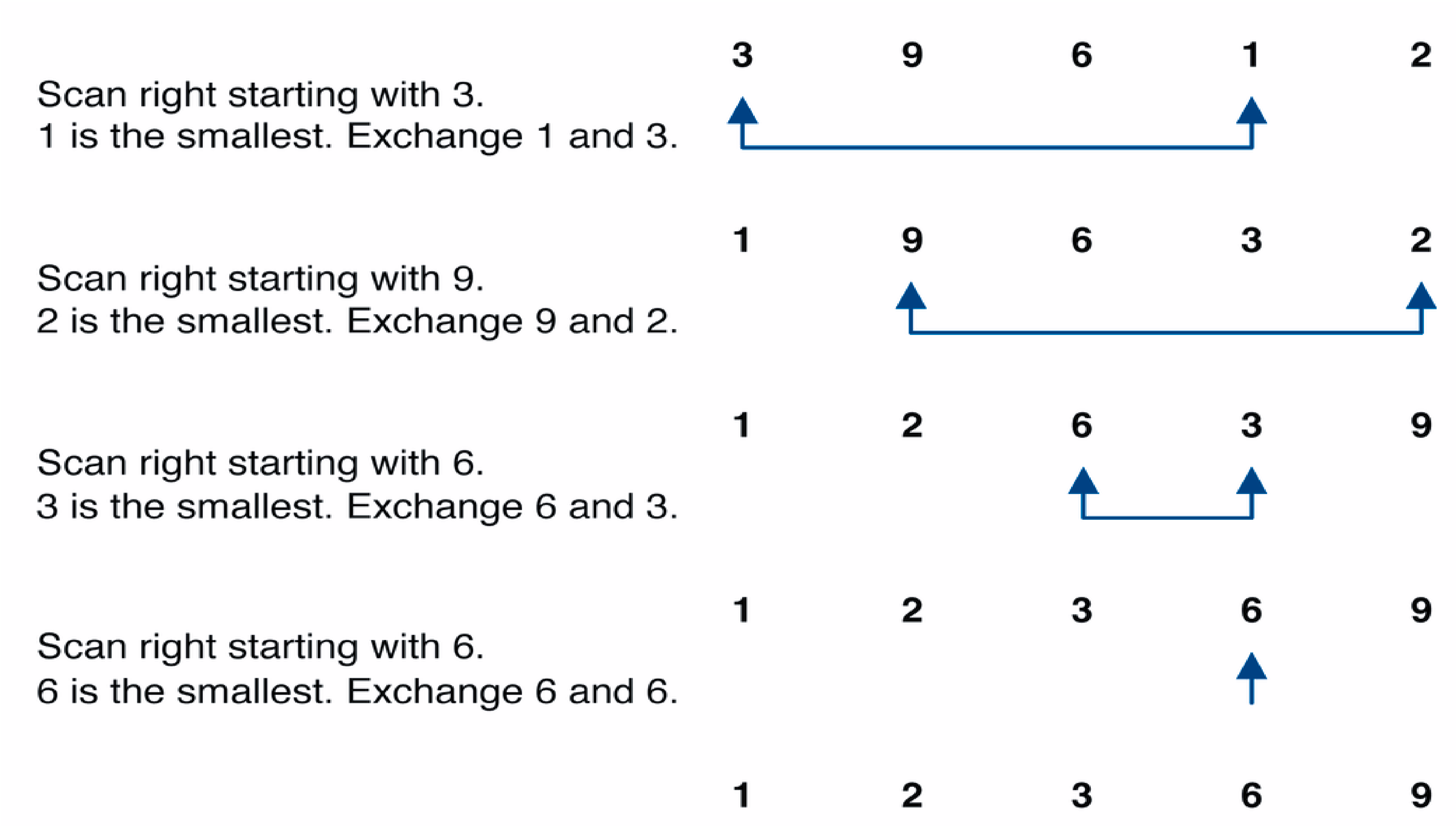
  - number of actual swaps performed depends on the data; out-of-order data performs many swaps

Selection sort

- **selection sort**: orders a list of values by repetitively putting a particular value into its final position

- more specifically:
  - find the smallest value in the list
  - swap it with the value in the first position
  - find the next smallest value in the list
  - swap it with the value in the second position
  - repeat until all values are in their proper places

Selection sort example



Selection sort example 2

Index	0	1	2	3	4	5	6	7
Value	27	63	1	72	64	58	14	9
1st pass	**1**	63	27	72	64	58	14	9
2nd pass	1	**9**	27	72	64	58	14	63
3rd pass	1	9	**14**	72	64	58	27	63
…								

Selection sort code

```
void selectionSort(___________________________) {
    for (int i = 0; ________________; ______________) {
        // find index of smallest element

        // Find index of ith smallest element

        // swap smallest element with a[i]
        swap(a, ______________, _____________);
    }
}
```

Selection sort code

```
void selectionSort(int a[], int n) {
    for (int i = 0; i < n; i++) {
        // Find index of ith smallest element
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[minIndex]) {
                minIndex = j;
            }
        }

        // swap smallest element with a[i]
        swap(a, i, minIndex);
    }
}
```

Selection sort runtime

- Running time for input size *n*:
  - In practice, a bit faster than bubble sort. Why?

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-1}(n-1-(i+1)+1)$$

$$=\sum_{i=0}^{n-1}(n-i-1)$$

$$=n\sum_{i=0}^{n-1}1-\sum_{i=0}^{n-1}i-\sum_{i=0}^{n-1}1$$

$$=n^2-\frac{(n-1)n}{2}-n$$

$$=\Theta(n^2)$$

Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

Insertion sort

- Simple sorting algorithm.
  - n-1 passes over the array
  - At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

	2	15	8	1	17	10	12	5
	0	1	2	3	4	5	6	7
after pass 2	2	8	15	1	17	10	12	5
	0	1	2	3	4	5	6	7
after pass 3	1	2	8	15	17	10	12	5
	0	1	2	3	4	5	6	7

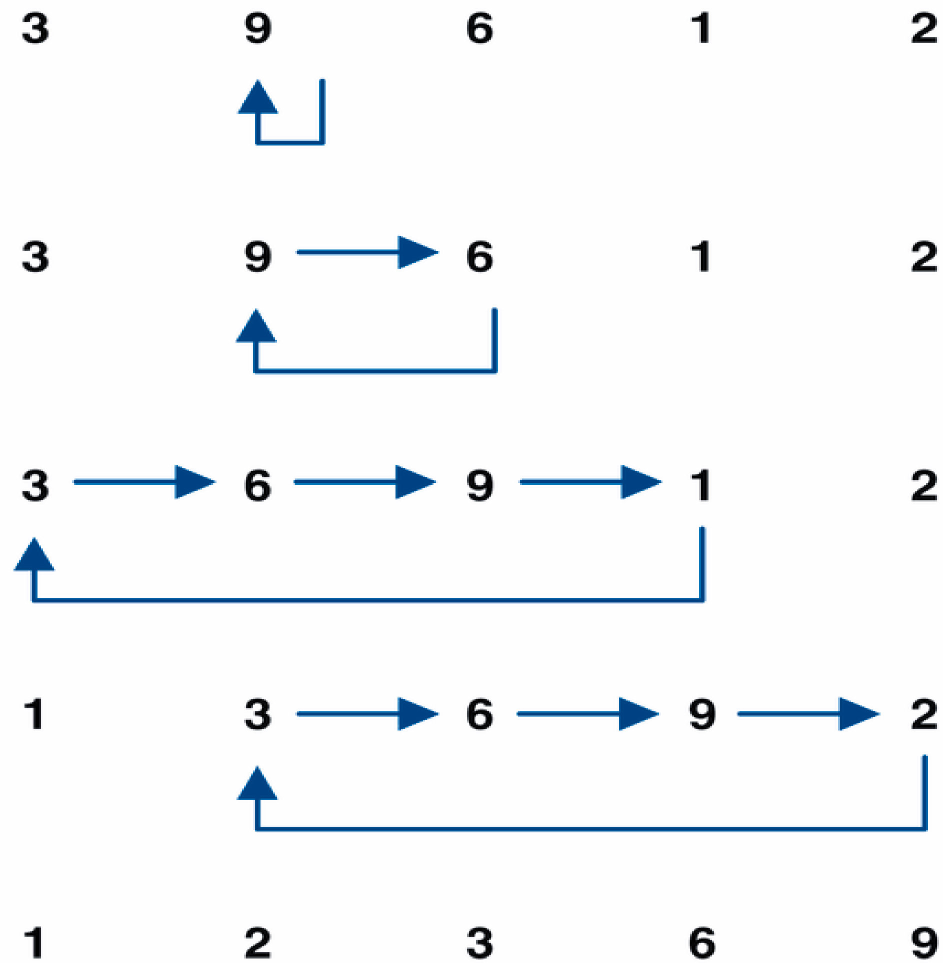
Insertion sort example

3 is sorted.
Shift nothing. Insert 9.

3 and 9 are sorted.
Shift 9 to the right. Insert 6.

3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.

1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.



Insertion sort code

```
void insertionSort(int[] a, int n) {
    int i, temp;
    for (i = 1; i < n; i++) {
        int temp = a[i];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
```

Insertion sort runtime

- worst case: reverse-ordered elements in array.

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2}$$

$$= \Theta(n^2)$$

- best case: array is in sorted ascending order.

$$\sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n)$$

- average case: each element is about halfway in order.

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2}(1 + 2 + 3 \ldots + (n-1)) = \frac{(n-1)n}{4}$$

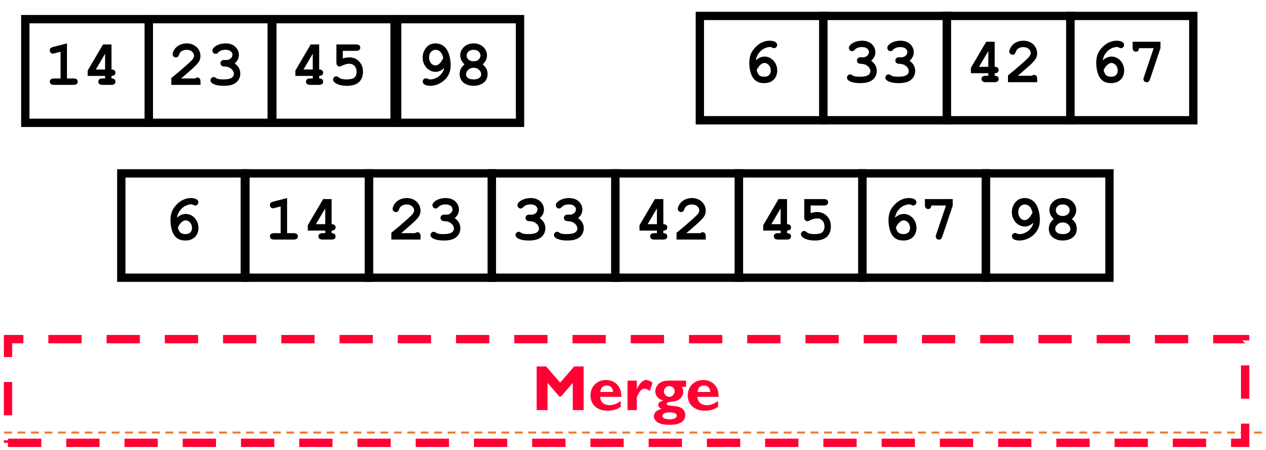
$$= \Theta(n^2)$$

Comparing sorts

- We've seen "simple" sorting algorithms so far, such as selection sort and insertion sort.
- They all use nested loops and perform approximately n^2 comparisons
- They are relatively inefficient

- So far we started with an unsorted array and directly sorted it
- We did not have any assumptions on the data
- What if we do some "preprocessing" of the input, that results in some sort of "interesting" or "useful" arrangements in the data, that helps in obtaining the final sorting?
- Let us see ..

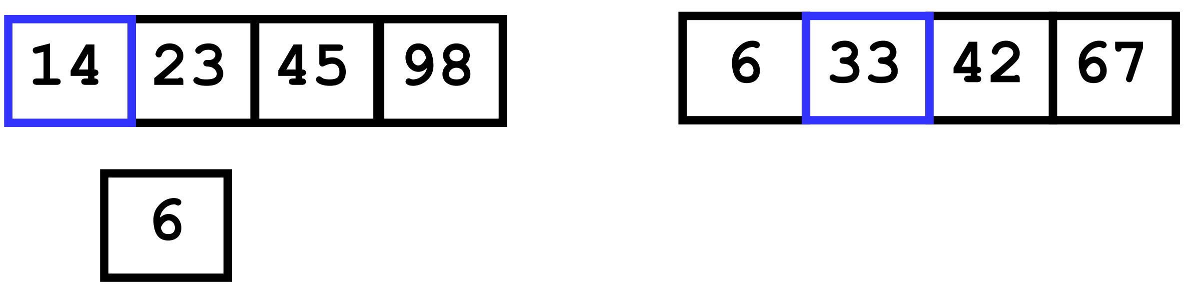
- Suppose we know that the input list/array is essentially a concatenation of two sorted subarrays.
  - Can we generate the final sorted array "quickly"?
- Suppose the subarrays are: A and B
- We want to sort the complete array, that is the concatenation of A and B
- We are allowed to use another temporary array C.


14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98
Merge

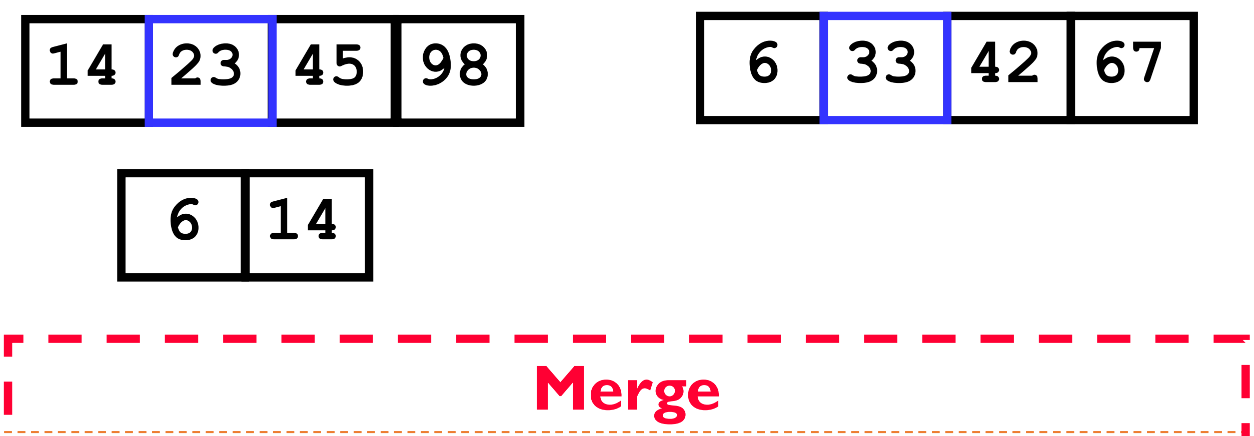
14	23	45	98

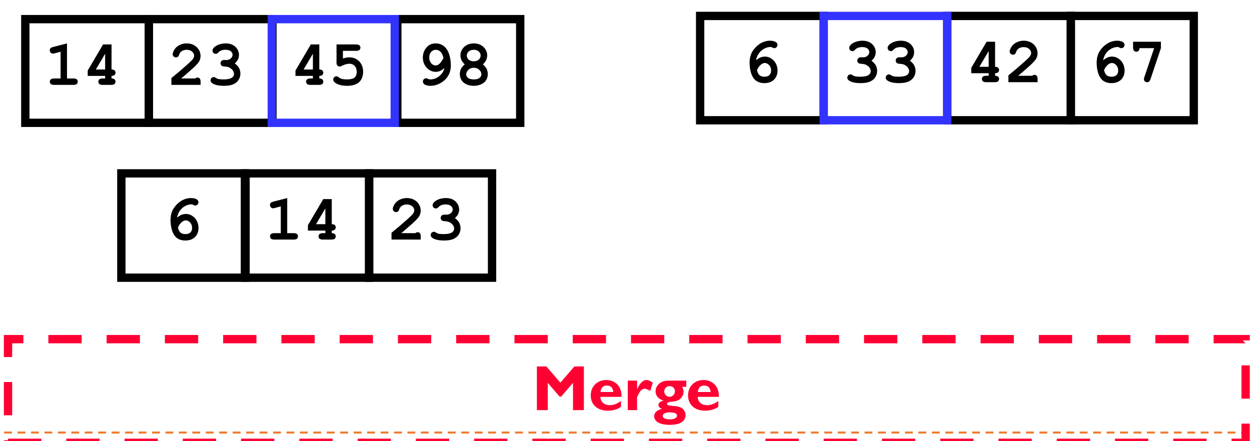
6	33	42	67

Merge



Merge


14 23 45 98
6 33 42 67
6 14
Merge

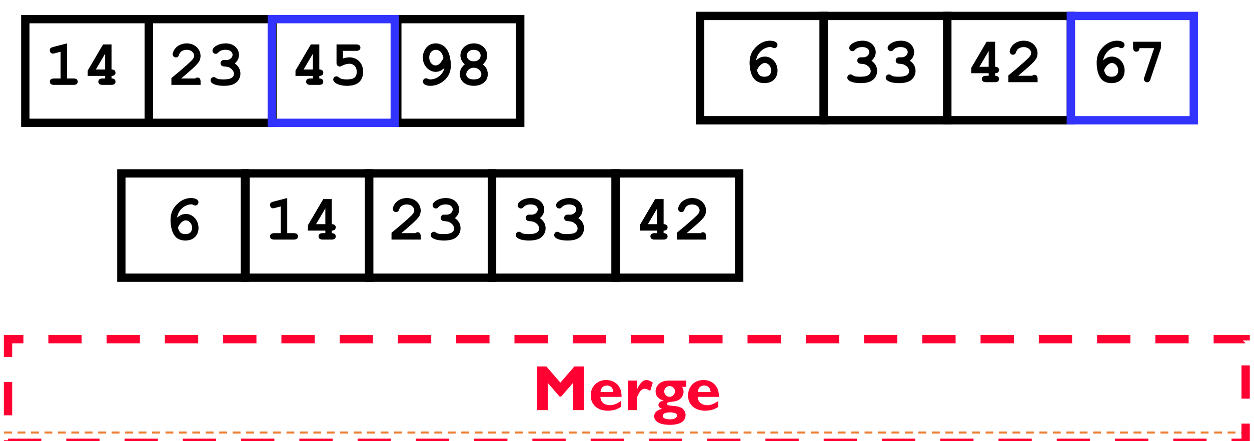

14 23 45 98
6 33 42 67
6 14 23
Merge

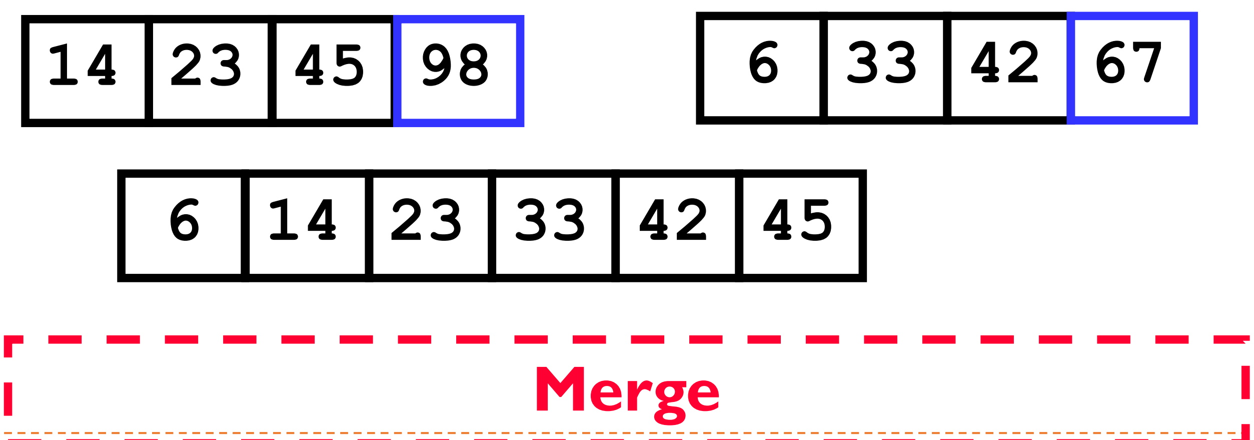
14	23	45	98

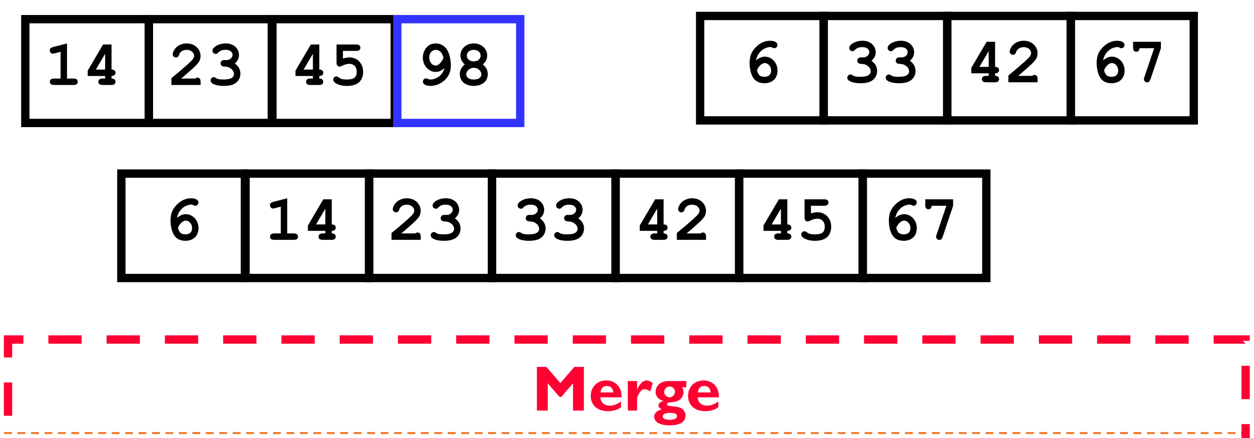
6	33	42	67

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

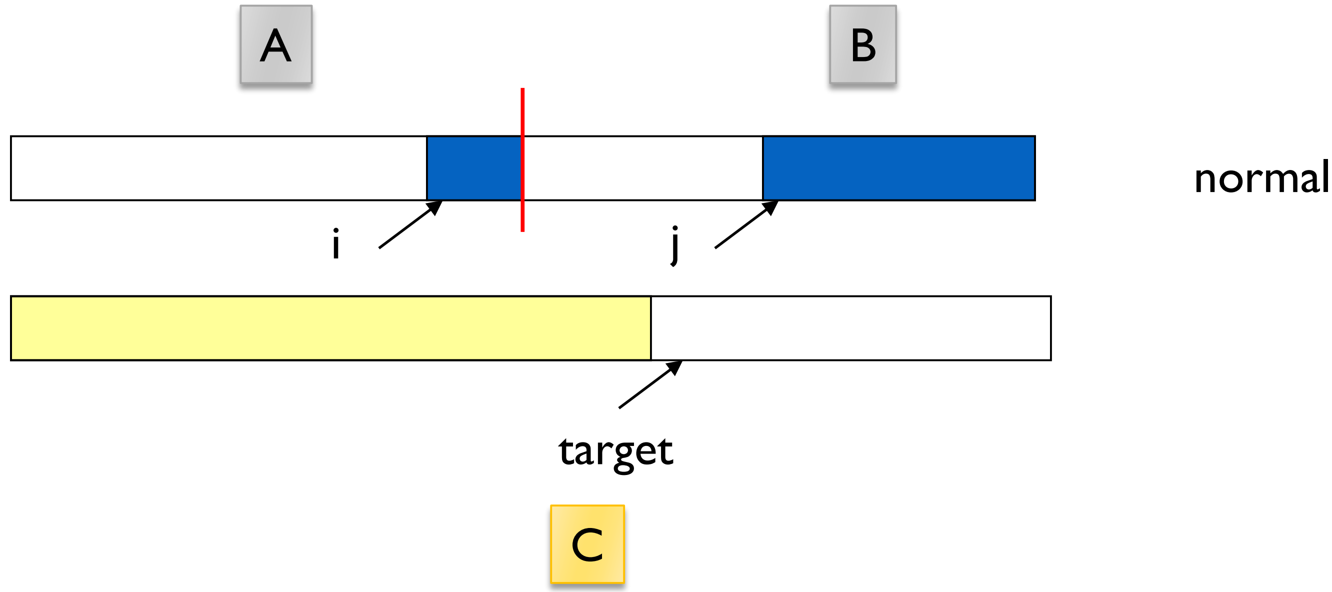
14	23	45	98

6	33	42	67

6	14	23	33	42	45	67	98

Merge

Merging

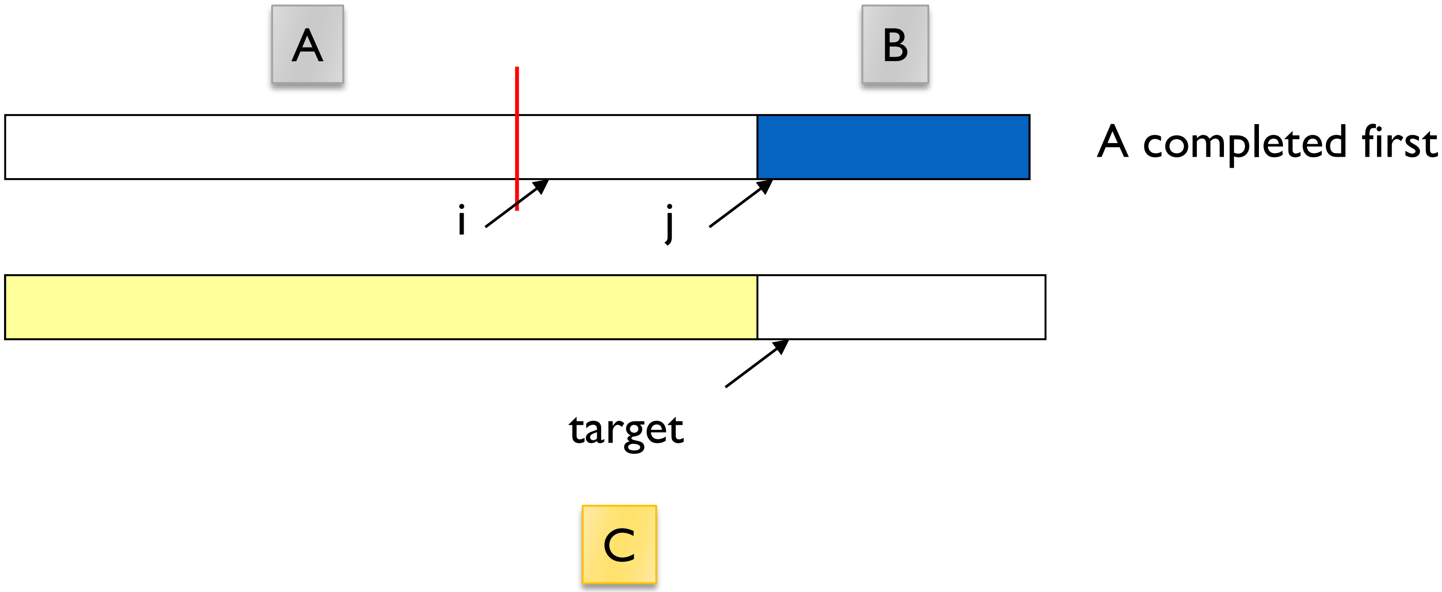



```
Loop    {
        if( A[i] < B[j] ) {

        } else {

        }
```

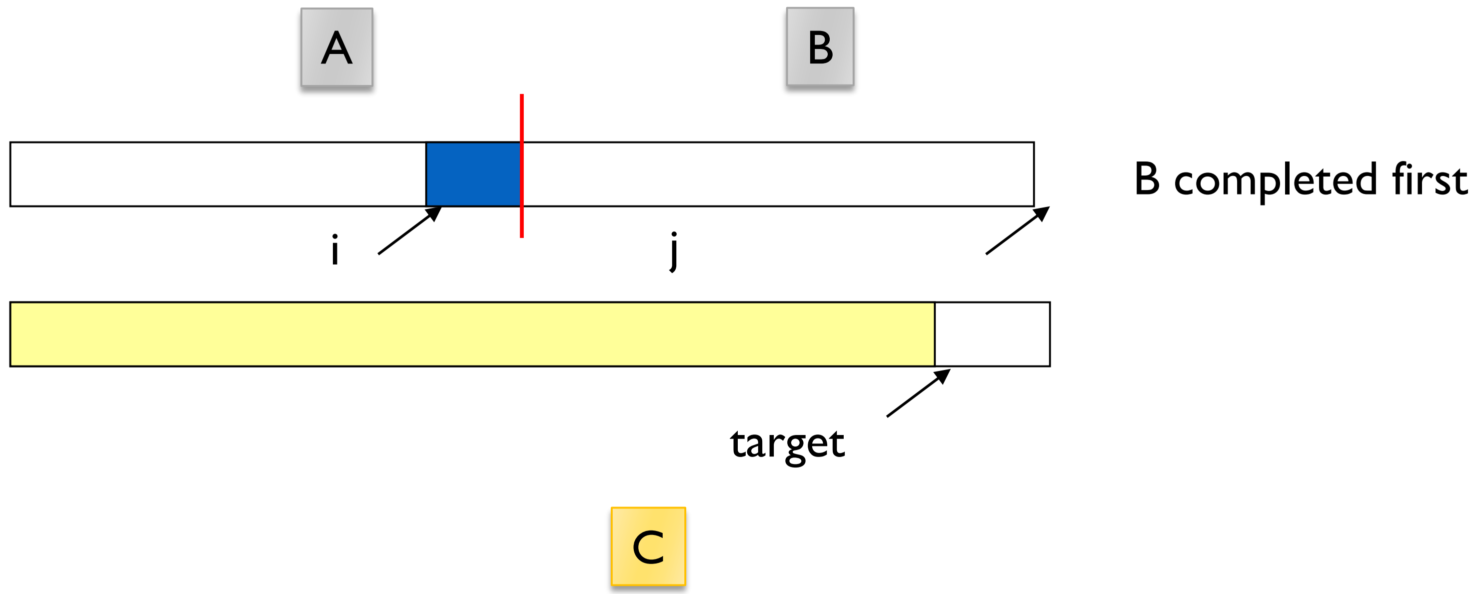
Merging




```
loop {

}
```

Merging




```
loop {

}
```

```
void merge(int A[], int m, int B[], int n, int C[])
{
        int i=0, j=0, k=0;
        while( _________________________________ )
        {
                if( A[i] < B[j] ) { C[k]=A[i]; k++; i++;}
                else { C[k]=B[j]; k++; j++; }
        }

        /*Put remaining elements of A into C*/
        while( ____________ ) {
                C[k]=A[i]; k++; i++;
        }
        /*Put remaining elements of B into C*/
        while( _____________)
                 { C[k]=B[j]; k++; j++; }
}
```

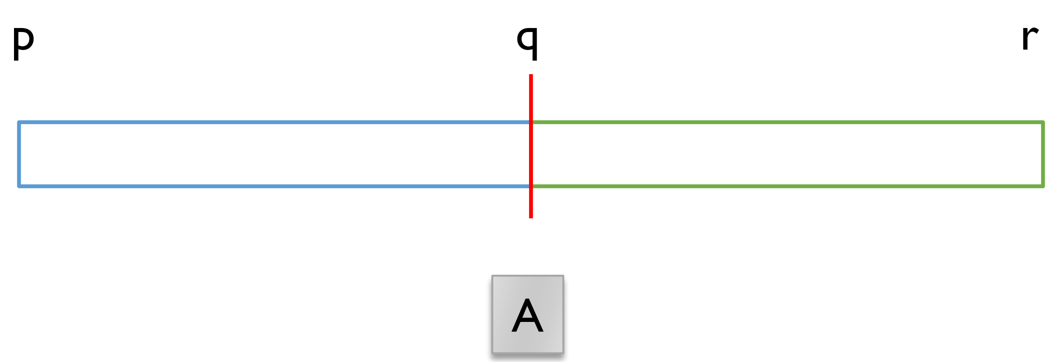
```
void merge(int A[], int m, int B[], int n, int C[])
{
        int i=0, j=0, k=0;
        while( (i < m) && (j < n) )
        {
                if( A[i] < B[j] ) { C[k]=A[i]; k++; i++;}
                else { C[k]=B[j]; k++; j++; }
        }

        /*Put remaining elements of A into C*/
        while( i < m ) {
                C[k]=A[i]; k++; i++;
        }
        /*Put remaining elements of B into C*/
        while( j < n )
                 { C[k]=B[j]; k++; j++; }
}
```

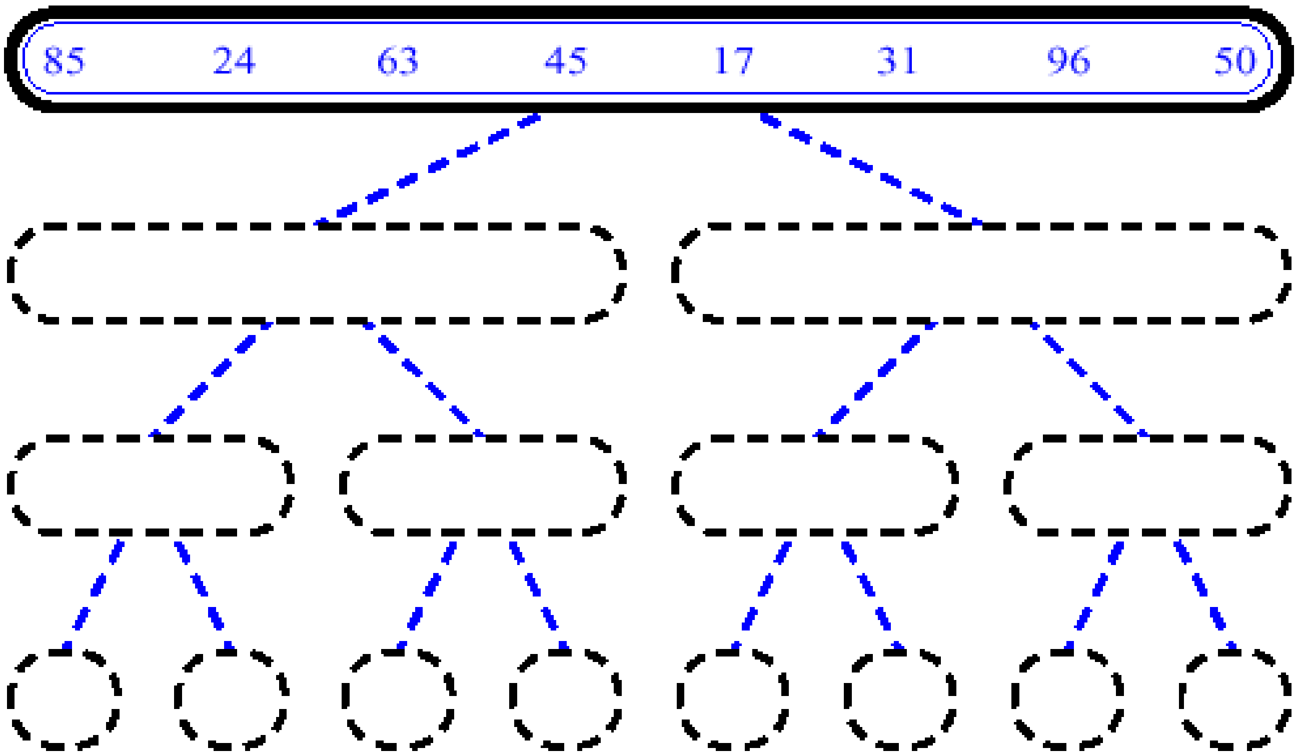
Time complexity?

- Time complexity?

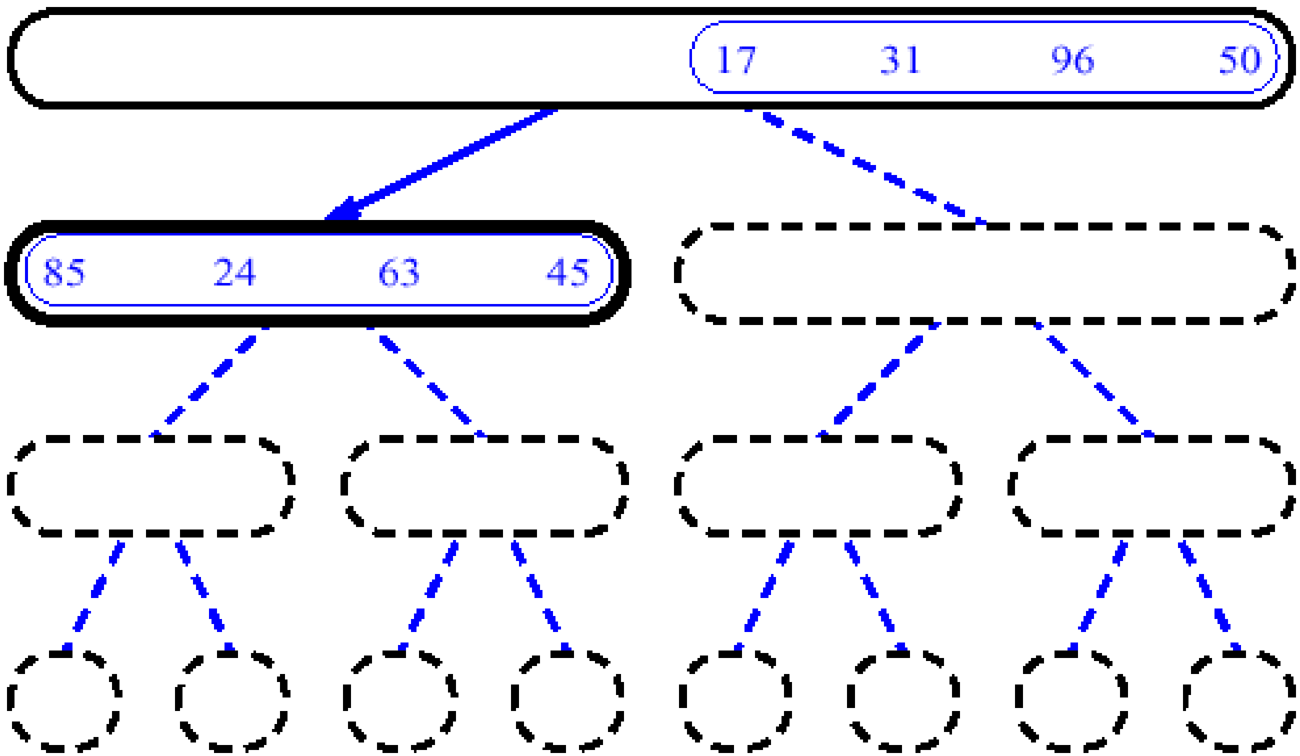
- Easy to adapt the code to the scenario where
  - There are no two separate arrays A and B
  - But one single array, which can be logically divided into two consecutive parts, and both parts are sorted part is sorted
  - The corresponding merge method needs to know which index in the array marks the last element of the first array (or firsr element of the second array)



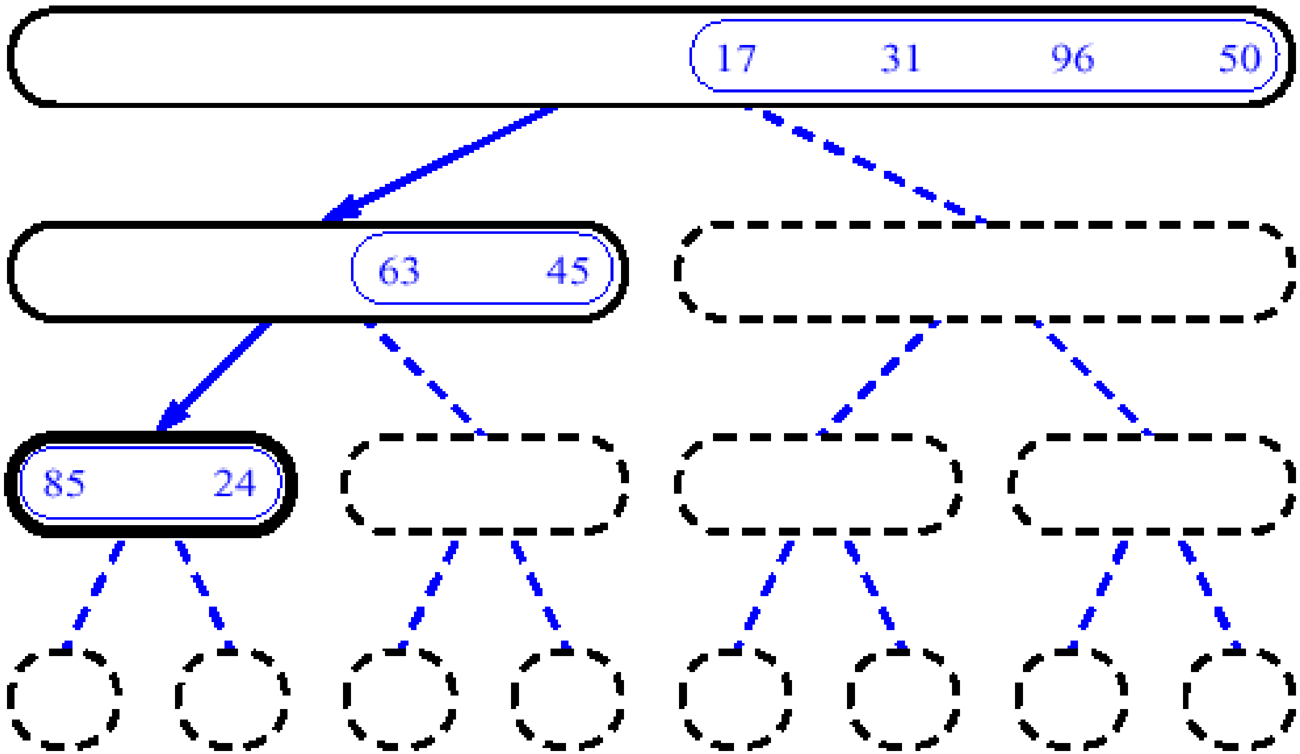
MergeSort (Example) - 1



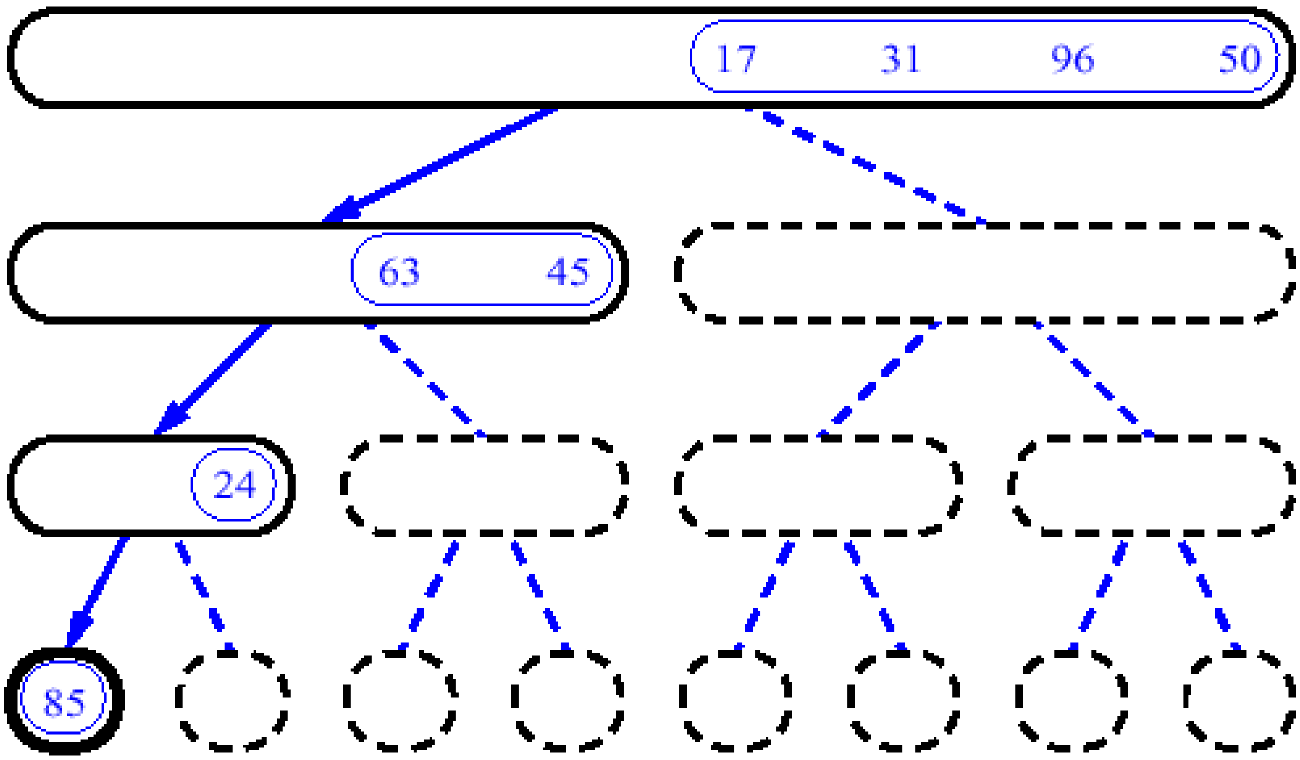
MergeSort (Example) - 2



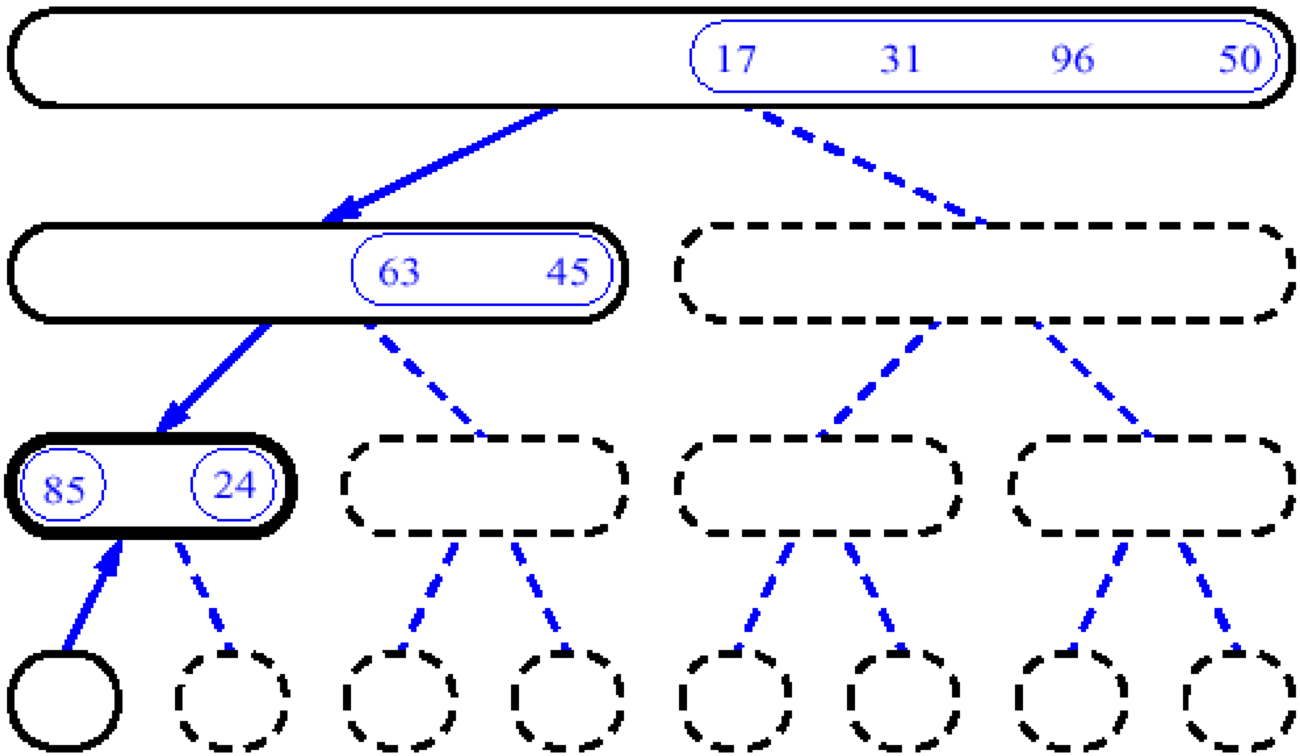
MergeSort (Example) - 3



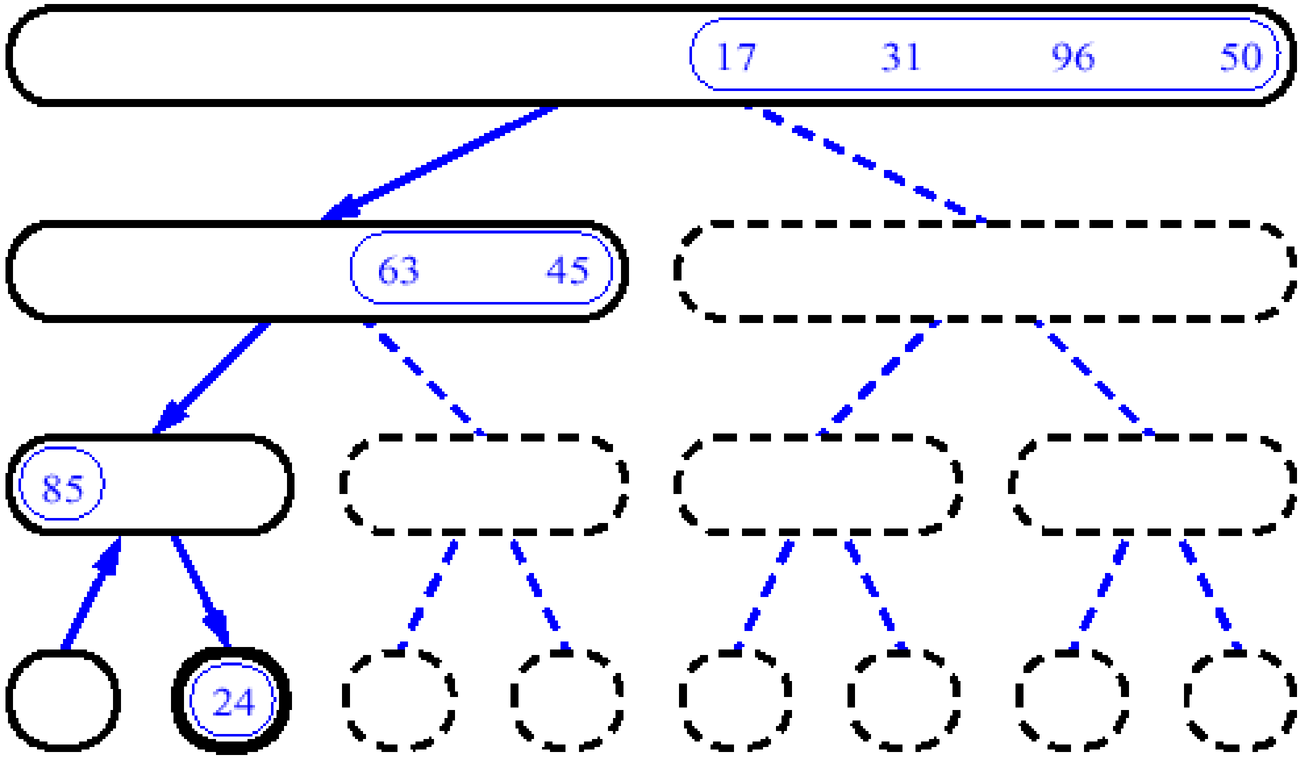
MergeSort (Example) - 4



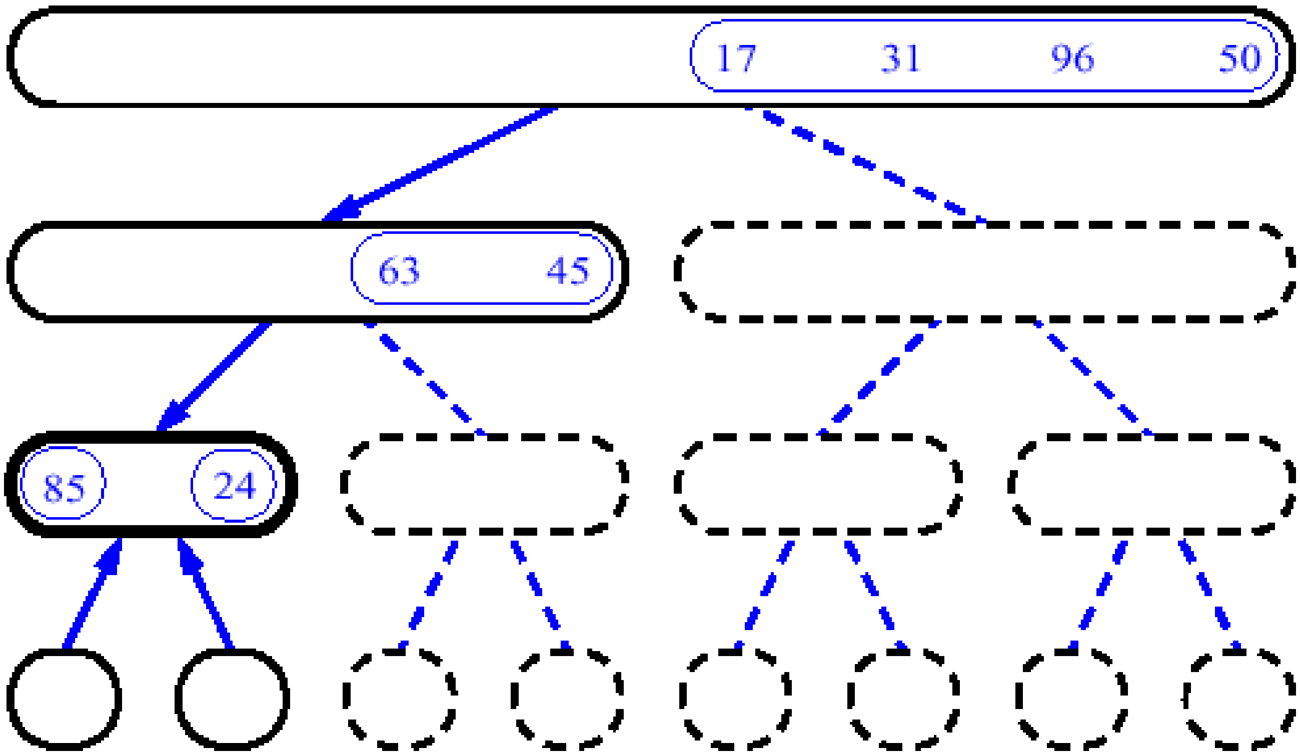
MergeSort (Example) - 5



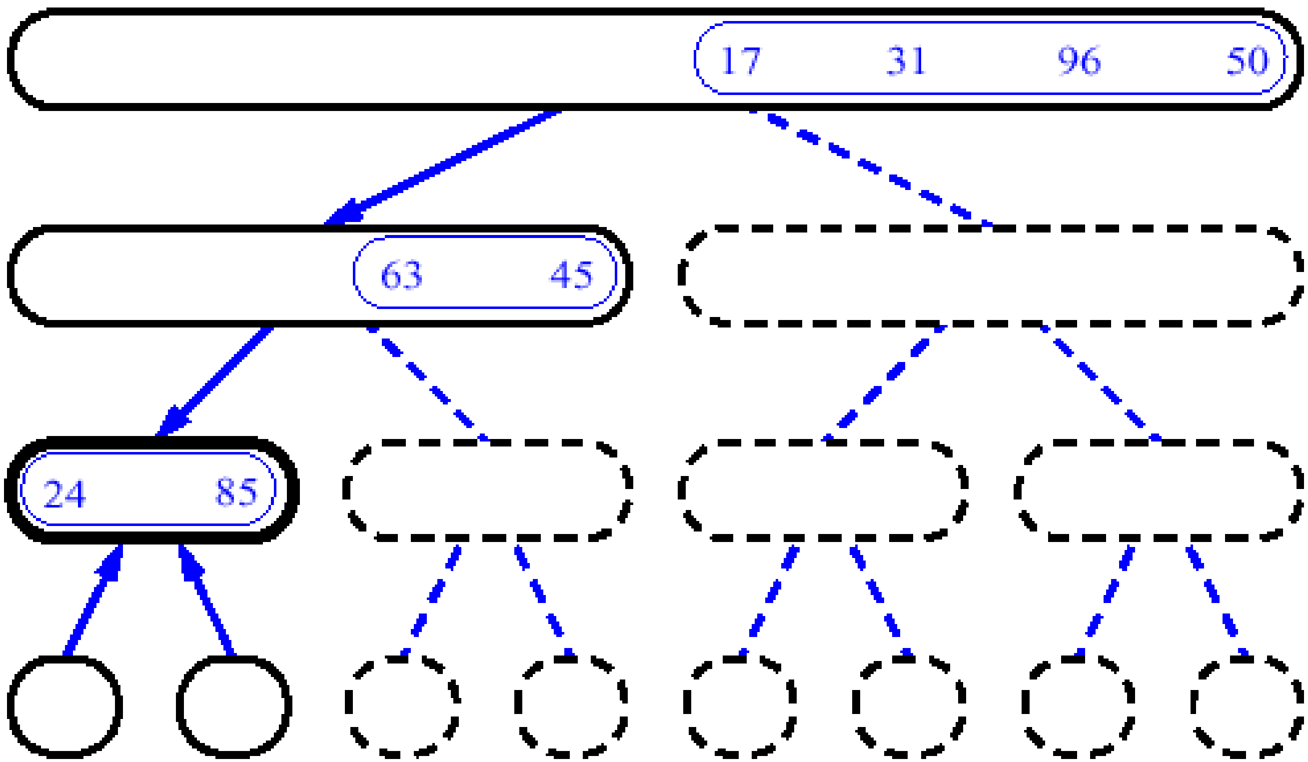
MergeSort (Example) - 6



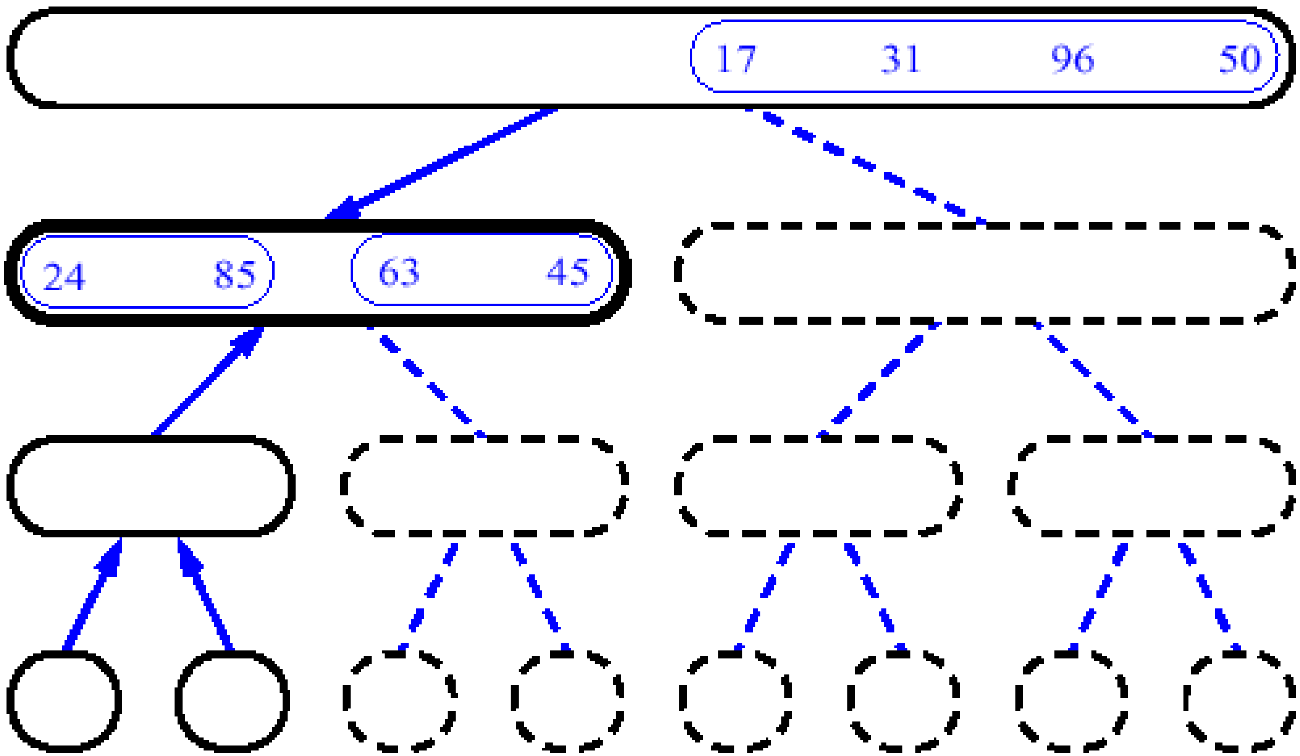
MergeSort (Example) - 7



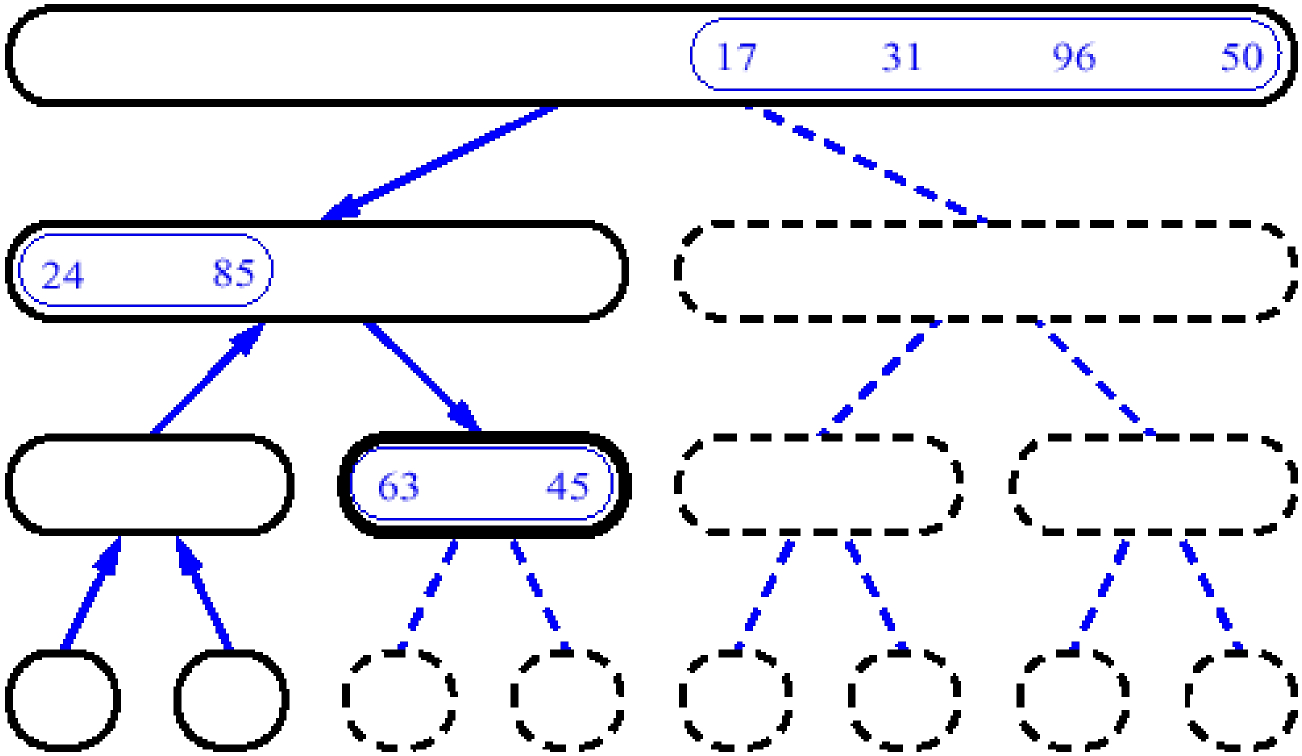
MergeSort (Example) - 8



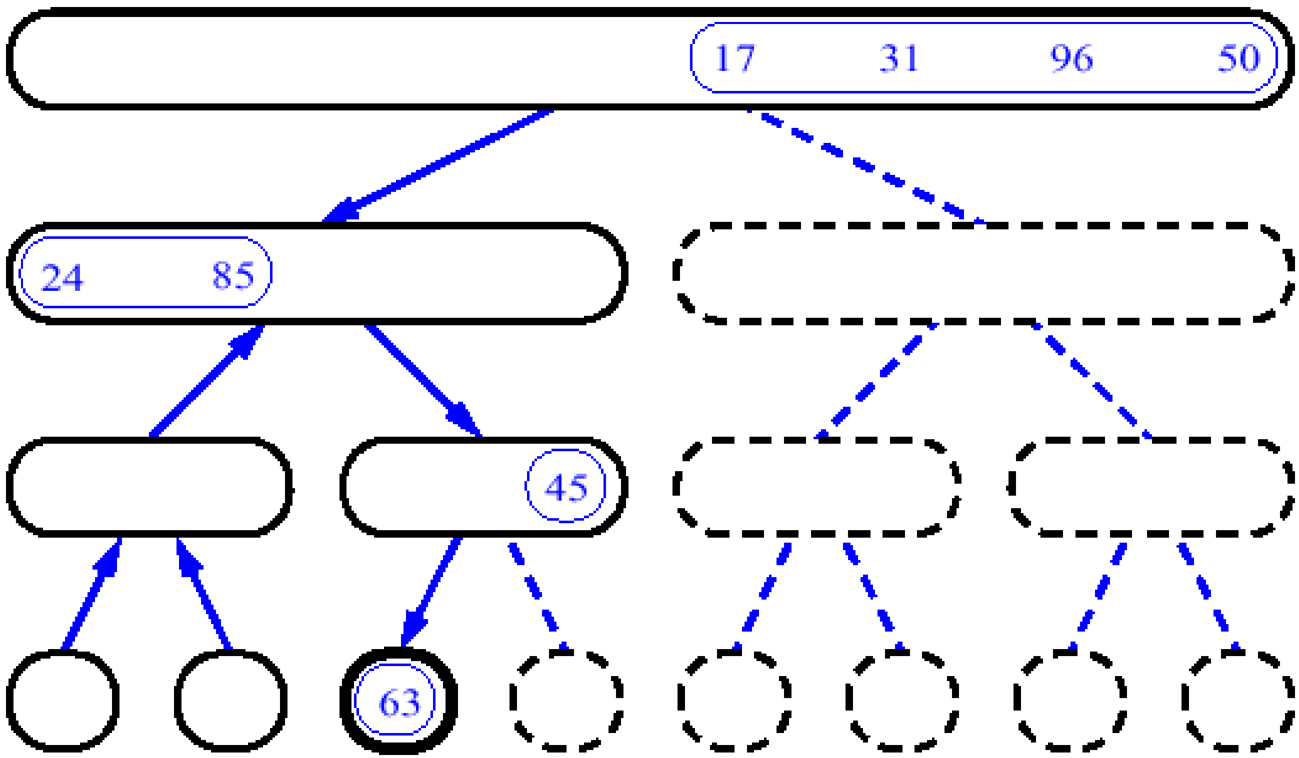
MergeSort (Example) - 9



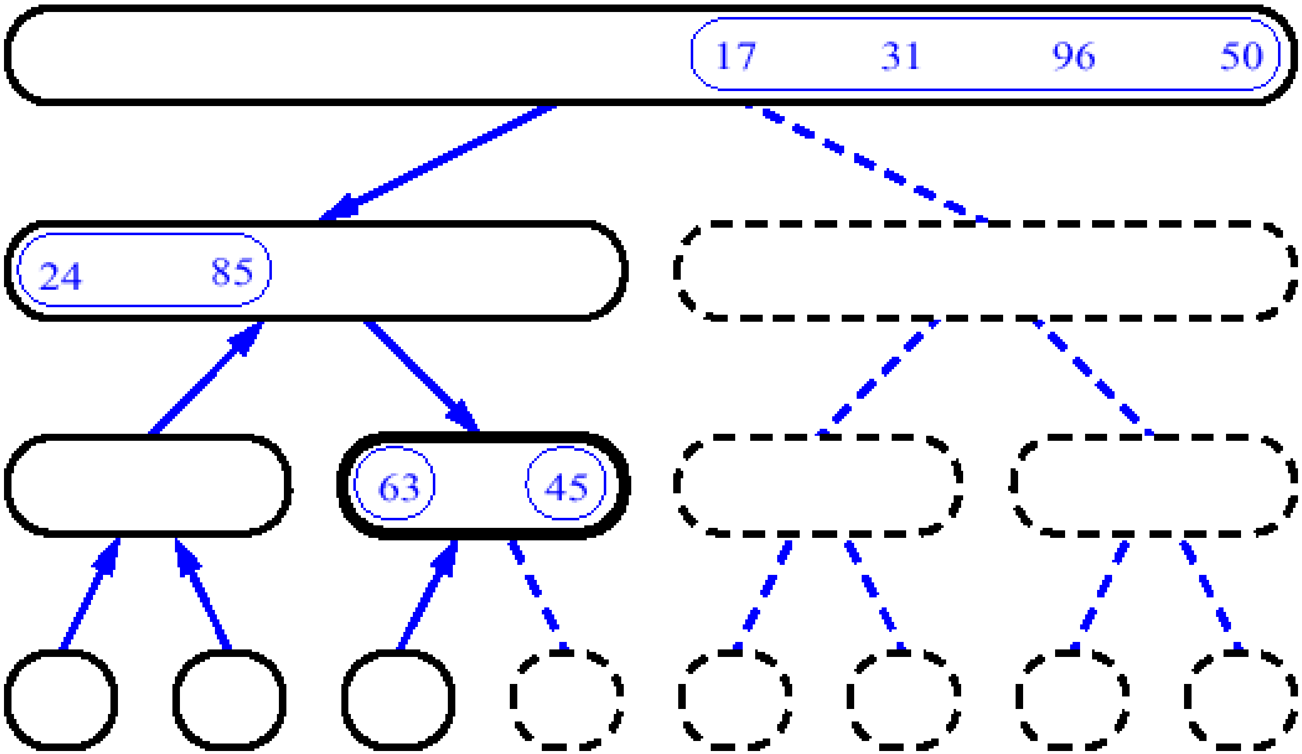
MergeSort (Example) - 10



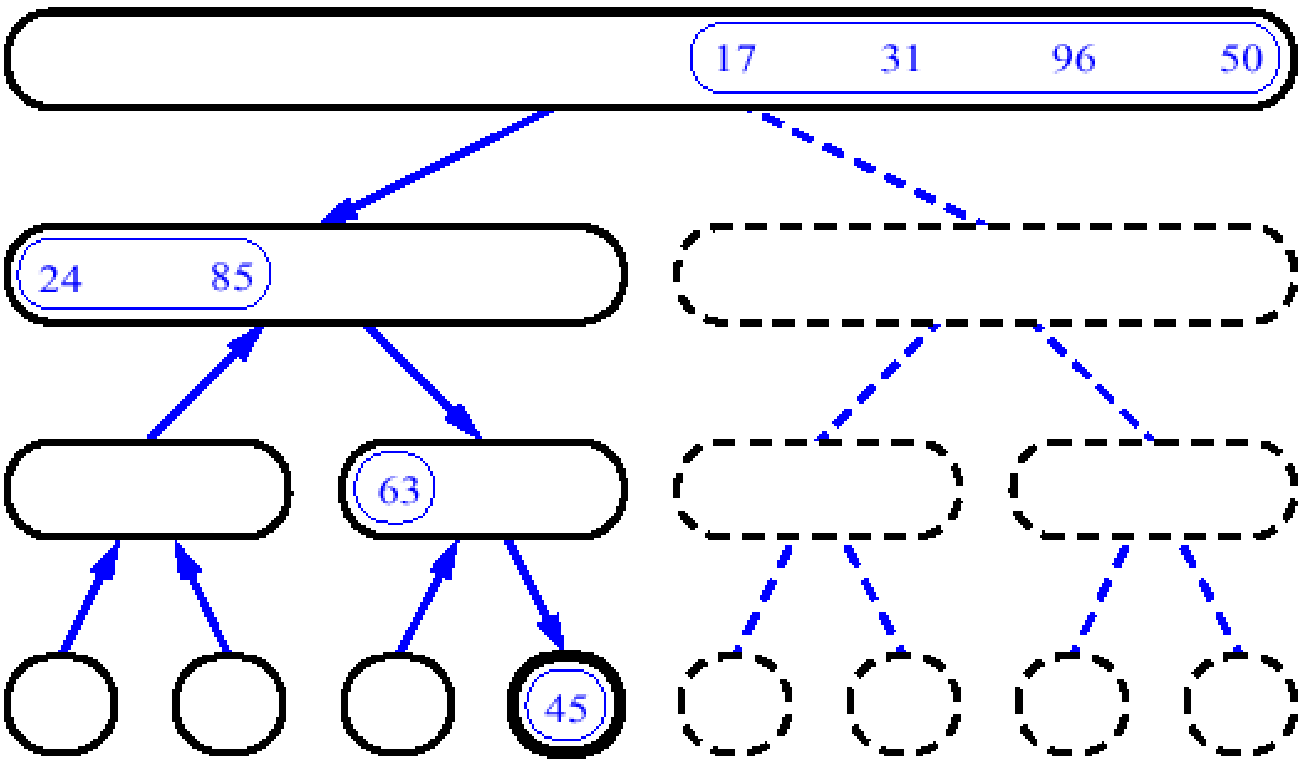
MergeSort (Example) - 11



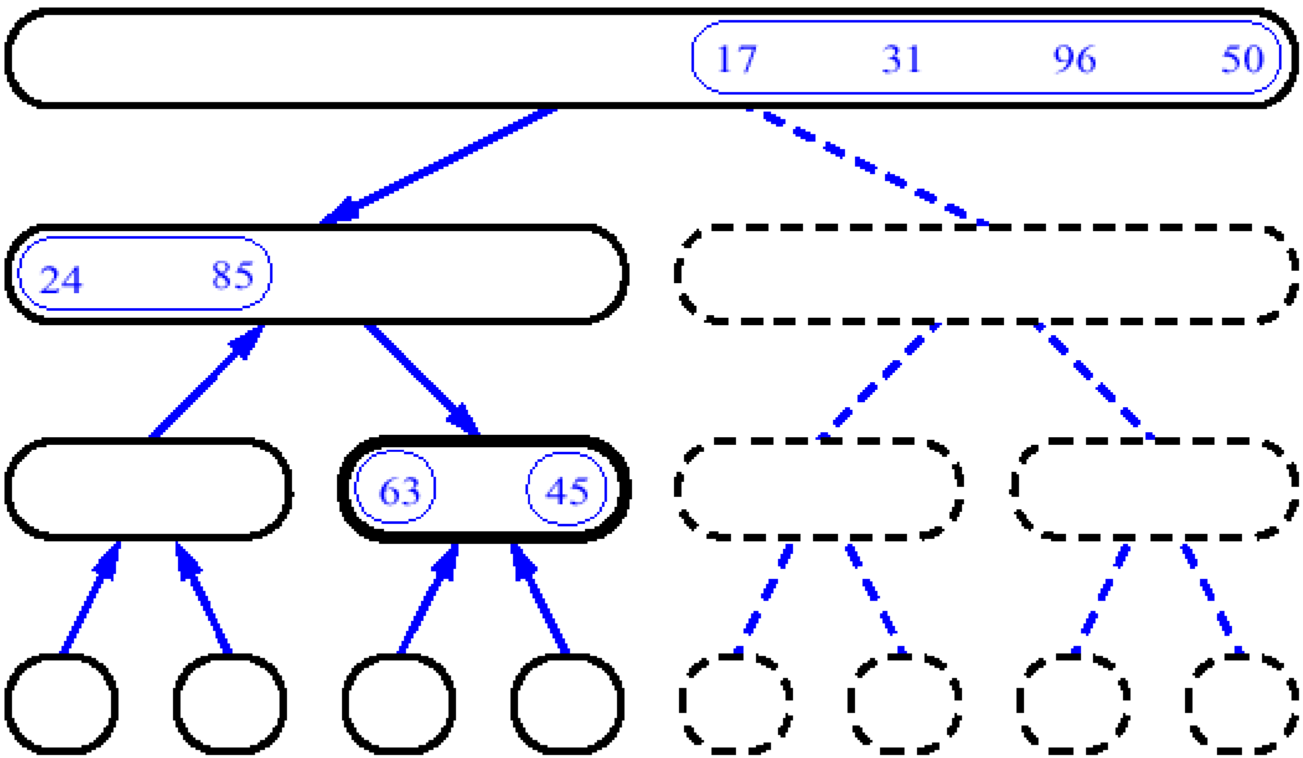
MergeSort (Example) - 12



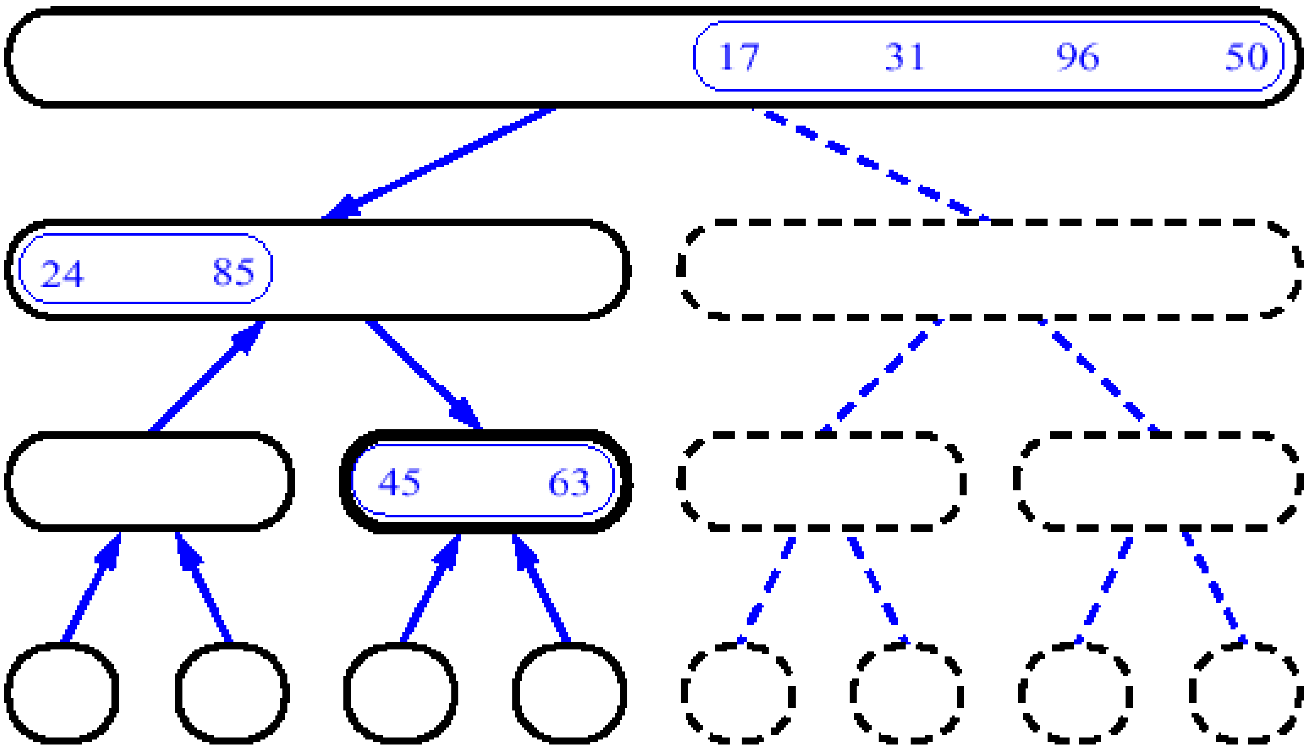
MergeSort (Example) - 13



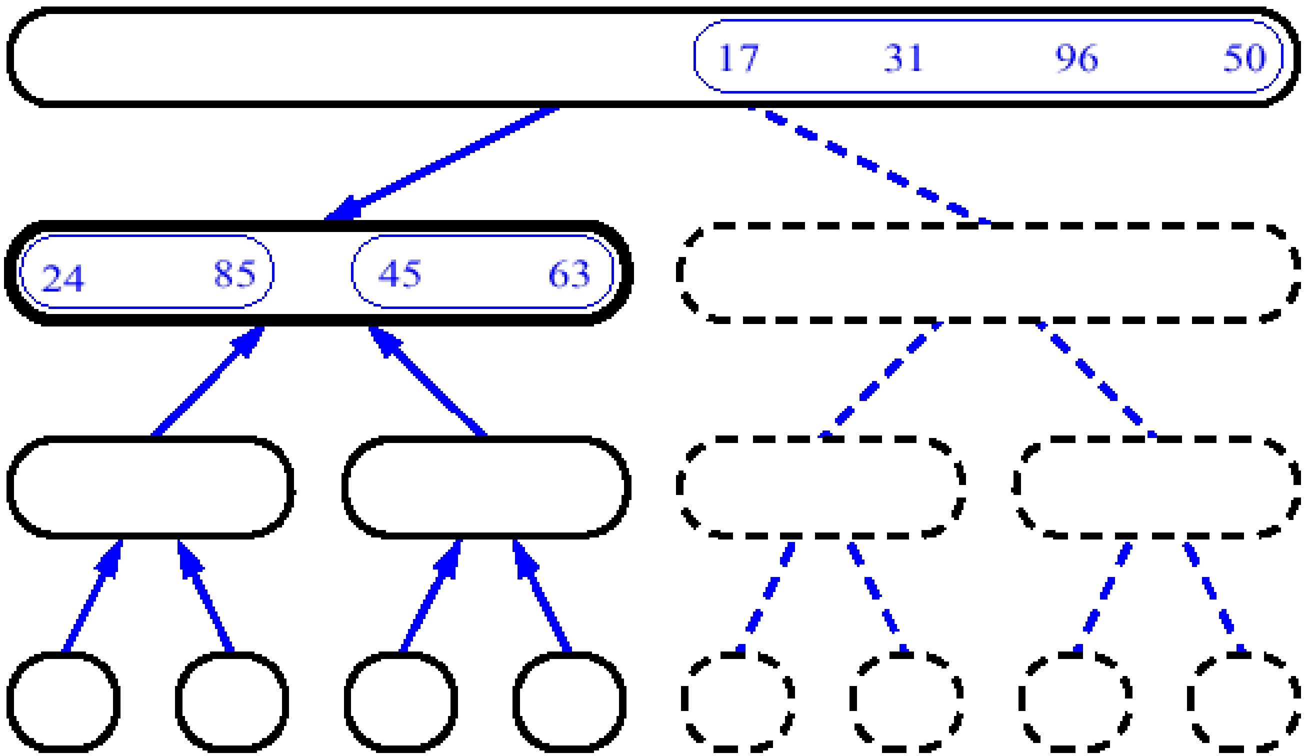
MergeSort (Example) - 14



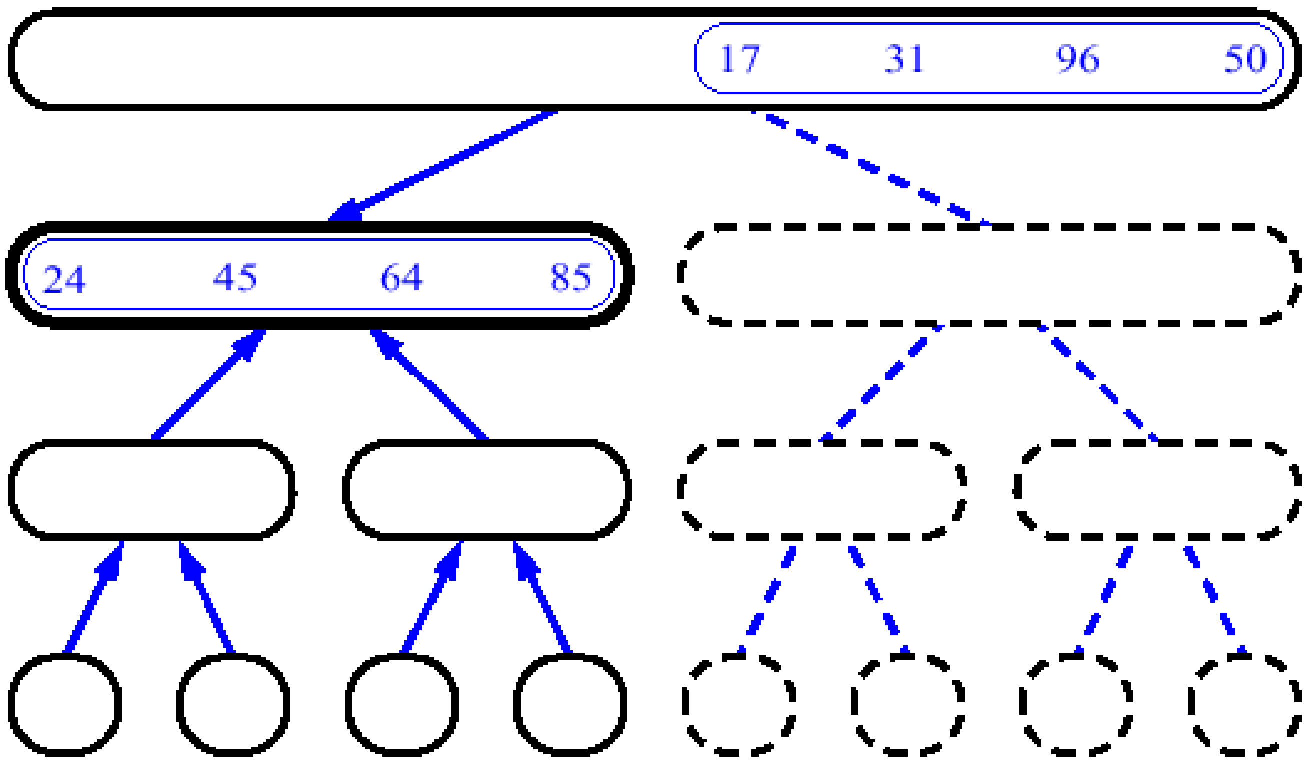
MergeSort (Example) - 15



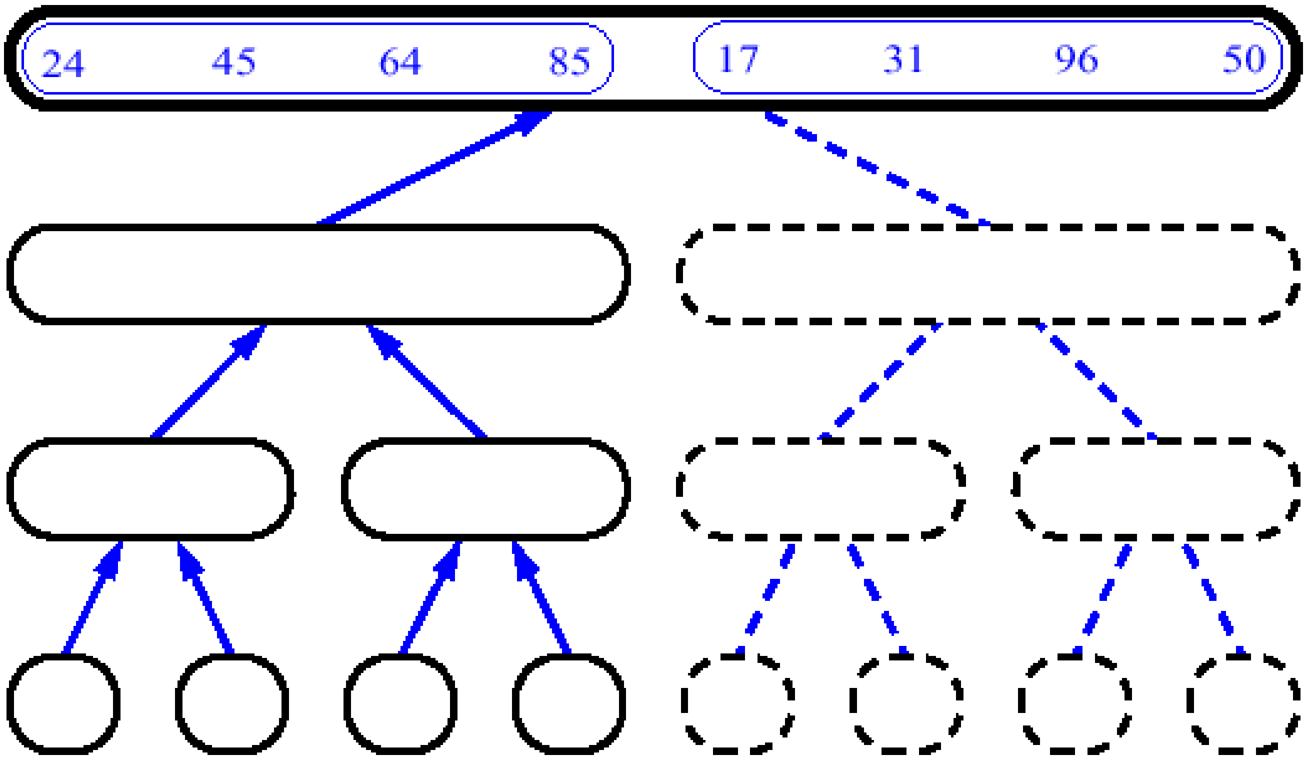
MergeSort (Example) - 16



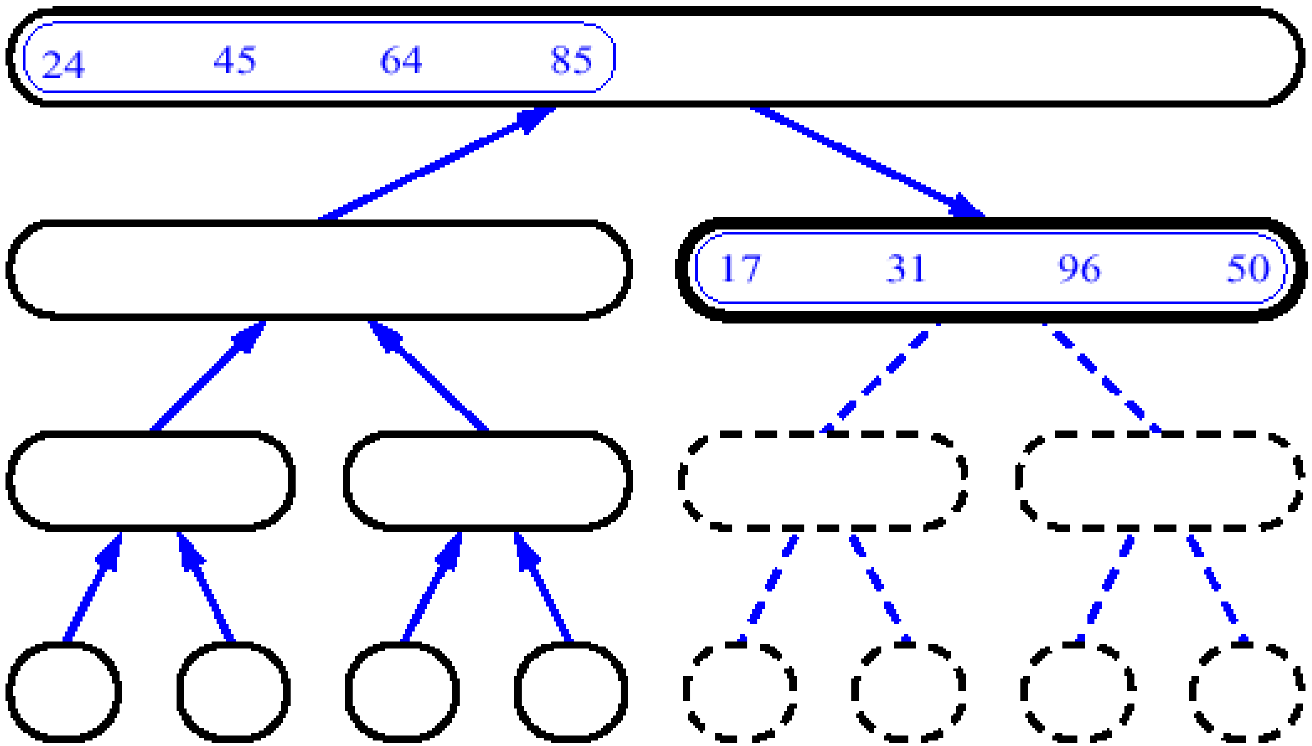
MergeSort (Example) - 17



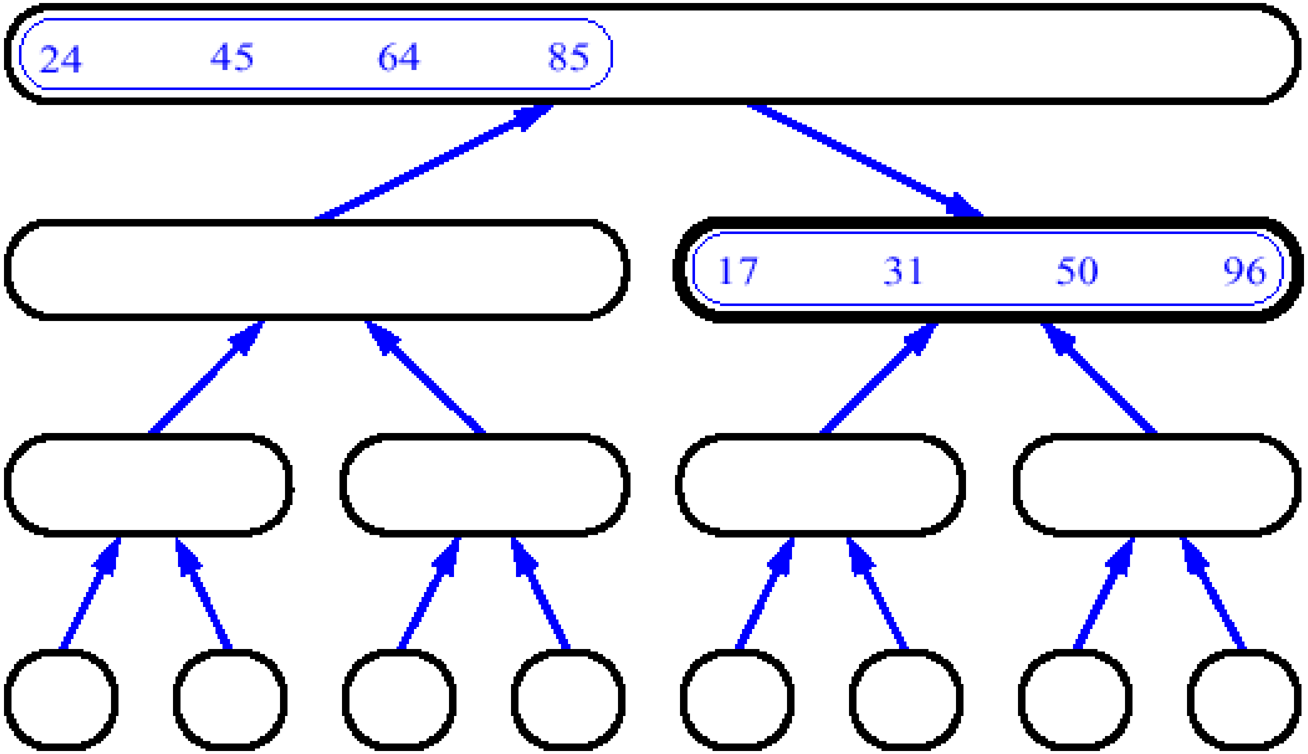
MergeSort (Example) - 18



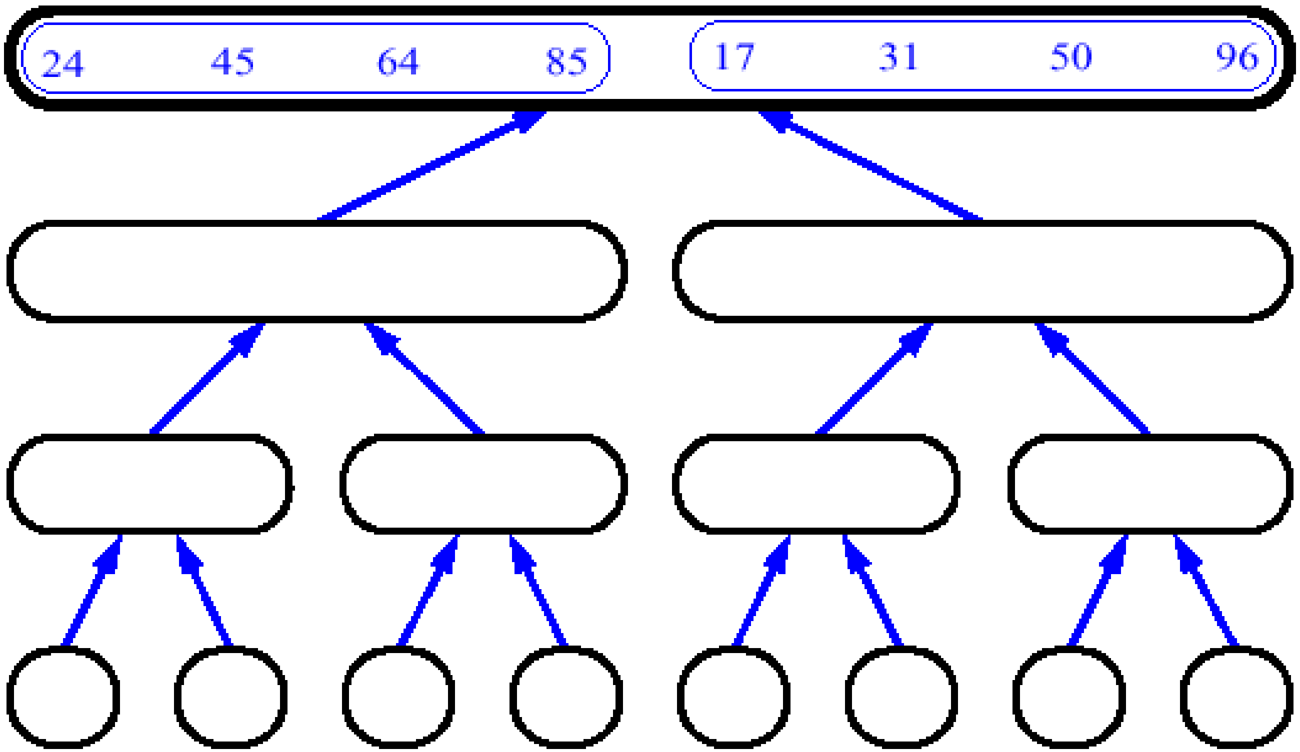
MergeSort (Example) - 19



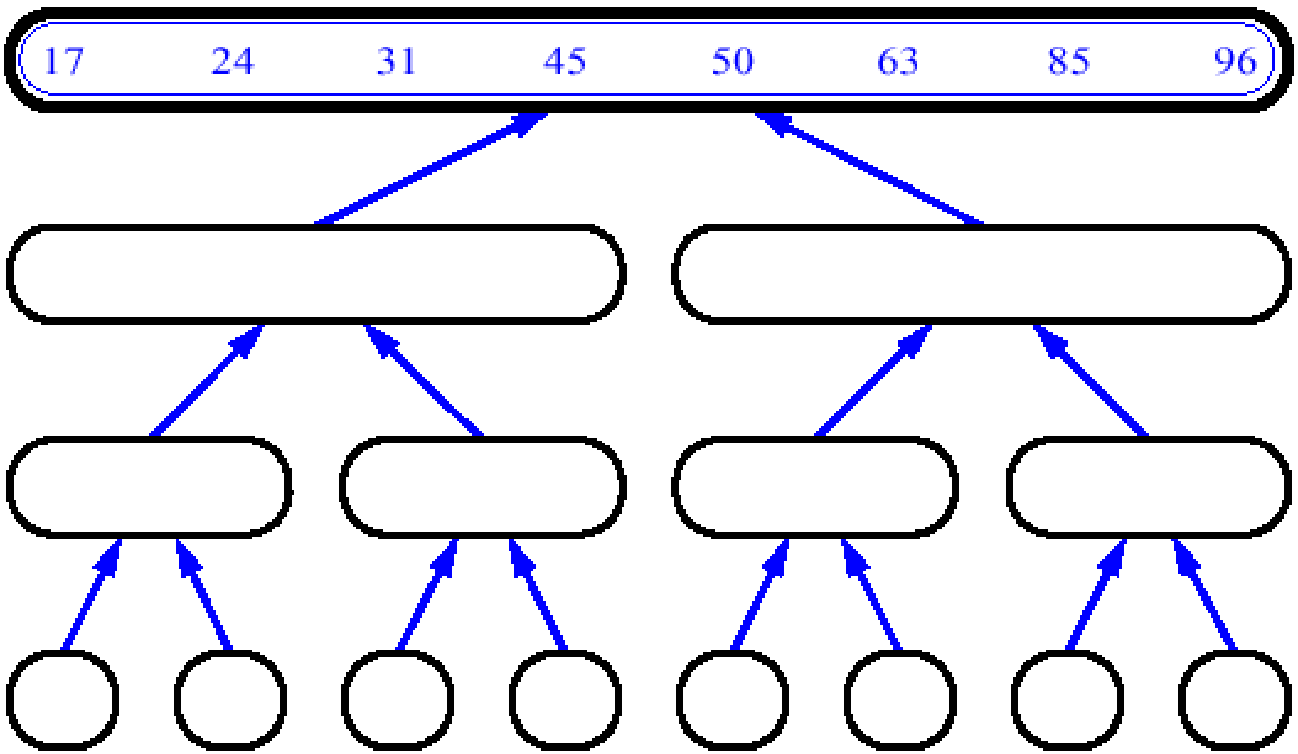
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22



Merge Sort

0	1	2	3	4	5	6	7
5	2	4	7	1	3	2	6
p			q				r

Alg.: MERGE-SORT(A, p, r)

if p < r ▷ Check for base case

then $q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

MERGE-SORT(A, p, q) ▷ Conquer

MERGE-SORT(A, q + 1, r) ▷ Conquer

MERGE(A, p, q, r) ▷ Combine

- Initial call: MERGE-SORT(A, 0, n-1)

Time complexity?

Quick Sort

Formal Worst-Case Analysis of Quicksort

- $T(n)$ = worst-case running time

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

- Solution to this recurrence relation is $T(n) = O(n^2)$

Randomized PARTITION

Alg.: RANDOMIZED-PARTITION(A, p, r)

i ← RANDOM(p, r)

exchange A[p] ↔ A[i]

return PARTITION(A, p, r)

Analysis of Randomized Quicksort

Alg. : RANDOMIZED-QUICKSORT(A, p, r)

The running time of Quicksort is dominated by PARTITION !!

if p < r

then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q - 1)

RANDOMIZED-QUICKSORT(A, q + 1, r)

For this and the subsequent slides, the reference is Cormen's book.

PARTITION

Alg.: PARTITION(A, p, r)

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

do if $A[j] \leq x$

then $i \leftarrow i + 1$

exchange $A[i] \leftrightarrow A[j]$

exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$

O(1) - constant

# of comparisons: X_k between the pivot and the other elements

O(1) - constant

Amount of work at call k: $c + X_k$

Average-Case Analysis of Quicksort

- Let X = **total number of comparisons performed in <u>all calls</u> to PARTITION:** $X = \sum_k X_k$

- The total work done over the **entire** execution of Quicksort is

  O(nc+X)=O(n+X)

- Need to estimate E(X)

Average-Case Analysis of Quicksort

- Let X = **total number of comparisons performed in all calls to PARTITION:** $X = \sum_k X_k$

- The total work done over the **entire** execution of Quicksort is

  O(n+X)

- Need to estimate E(X)

Notation

z_7	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_2
7	9	8	3	5	4	1	6	10	2

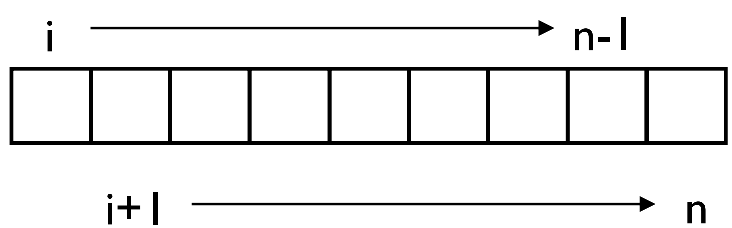
- Rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the *i*-th smallest element
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ the set of elements between z_i and z_j, inclusive

Total Number of Comparisons in PARTITION

- Define X_{ij} = I {z_i is compared to z_j }

- Total number of comparisons X performed by the algorithm:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$



Expected Number of Total Comparisons in PARTITION

- Compute the **expected value of X:**

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E\left[X_{ij}\right] =$$

by linearity
of expectation

indicator
random variable

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

the expectation of X_{ij} is equal to the probability of the event "z_i is compared to z_j"

Comparisons in PARTITION : Observation 1

- Each pair of elements is compared **at most once** during the entire execution of the algorithm
  - Elements are compared only to the pivot point!
  - Pivot point is excluded from future calls to PARTITION

Comparisons in PARTITION: Observation 2

- Only the pivot is compared with elements in both partitions!

z_7	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_2
7	9	8	3	5	4	1	6	10	2

$Z_{1,6} = \{1, 2, 3, 4, 5, 6\}$ $\{7\}$ $Z_{8,10} = \{8, 9, 10\}$

pivot

Elements between different partitions are <u>never</u> compared!

Comparisons in PARTITION

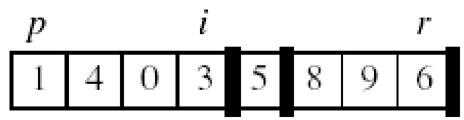
z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

$Z_{1,6} = \{1, 2, 3, 4, 5, 6\}$ $\{7\}$ $Z_{8,10} = \{8, 9, 10\}$

$$\Pr\{z_i \textit{ is compared to } z_j\}?$$

- Case 1: pivot chosen such as: $z_i < x < z_j$
  - z_i and z_j will never be compared
- Case 2: z_i or z_j is the pivot
  - z_i and z_j will be compared
  - only if one of them is chosen as pivot before any other element in range z_i to z_j

See why ☺



Z2 (0) will never be compared with z6 (6) since z5 (element 5 which belongs to [z_2, z_6]) was chosen as a pivot first !

Probability of comparing z_i with z_j

Pr{ z_i is compared to z_j } =

Pr{ z_i is the first pivot chosen from Z_{ij} } +

Pr{ z_j is the first pivot chosen from Z_{ij} }

$= 1/(j - i + 1) + 1/(j - i + 1) = 2/(j - i + 1)$

- There are $j - i + 1$ elements between z_i and z_j
  - Pivot is chosen randomly and independently
  - The probability that any particular element is the first one chosen is $1/(j - i + 1)$

Number of Comparisons in PARTITION

Expected number of comparisons in PARTITION:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n)$$

(set k=j-i) (harmonic series)

$$= O(n \lg n)$$

$\Rightarrow$ Expected running time of Quicksort using RANDOMIZED-PARTITION is $O(n\lg n)$