Fundamentals of Data Structures

S. Y. B. Tech CSE Semester – III

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

Searching

- Searching is the process of determining whether or not a given value exists in a data structure or a storage media.
- Two searching methods are: **linear search and binary search**.

- The linear (or sequential) search algorithm on an array is:

  1. Sequentially scan the array, comparing each array item with the searched value.

  2. If a match is found; return the index of the matched element; otherwise return –1.

Searching

- When we maintain a collection of data, one of the operations we need is a search routine to locate desired data quickly.
- Here's the problem statement:

  Given a value X, return the index of X in the array, if such X exists. Otherwise, return NOT_FOUND (-1). We assume there are no duplicate entries in the array.

- We will count the number of comparisons in the algorithms

  - The ideal searching algorithm will make the least possible number of comparisons to locate the desired data.

  – Two separate performance analysis are normally done, one for successful search and another for unsuccessful search.

Linear Search Algorithm

```
Algorithm Search(array,target,size)
  {

 for i = 1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i >= size)
     print("Target not found")
}
```

Scan the array

```
Algorithm Search(array,target,size)
   {

  for i=1 to n do
 {
      if(array[i] = target)
     {
       print("Target data found")
     break;
     }
 }
 if(i>size)
      print("Target not found")

 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i>=size)
     print("Target not found")

}
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i>=size)
     print("Target not found")

}
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
   {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i>=size)
     print("Target not found")
 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
{
      print("Target data found")
    break;
}
}
if(i>=size)
     print("Target not found")

}
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
   {

 for i=1 to n do
{
      if(array[i] = target)
{
       print("Target data found")
     break;
}
}
if(i>=size)
      print("Target not found")
  }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
   {

 for i=1 to n do
{
      if(array[i] = target)
{
       print("Target data found")
     break;
}
}
if(i>=size)
     print("Target not found")
 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n d
{
     if(array[i]
    {
      print(“Target data found”)
    break;
    }
}
if(i>=size)
     print(“Target not found”)
 }
```

Target data found

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break
    }
}
if(i>=size)
     print("Target not found")
 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

Linear Search Analysis: Best Case

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
     }
}
if(i>=size)
     print("Target not found")
  }
```

Best Case:
1 comparison

Best Case: match with the first item

target = 7

7	12	5	22	13	32

Linear Search Analysis: Worst Case

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
      if(array[i] = target)
     {
       print("Target data found")
     break;
      }
}
if(i>=size)
     print("Target not found")
 }
```

Worst Case: match with the last item (or no match)

7	12	5	22	13	32

target = 32

Worst Case:
N comparisons

Sequential search

```
int  seqsearch(int key)
{
    for i= 0 to n
     {
       if(key==a[i])
        {
          pos=i;
          flag=1;
          break;
        }
     }
   if(flag==1)
    return pos;
   else
  return -1;
}
```

```
class search
{
  int a[20],n;
  public:
   int seqsearch(int);
   void accept();
   void display();
};
```

Variations of Sequential Search

- **There are three such variations:**
  - **Sentinel search**

Sentinel search

- The algorithm ends either when the target is found or when the last element is compared
- The algorithm can be modified to eliminate the end of list test by placing the target at the end of list as just one additional entry
- This additional entry at the end of the list is called as *sentinel*

Sentinel search

```
int  seqsearch_sentinel(int key)
{
   a[n]=key;   //place target at the end of the list
  While (key!=a[i])
    {
          i++;
    }

   if(i<n)
    return i;
   else
    return -1;   //not found
}
```

- a) Write C++ program to store roll numbers of student in array who attended training program in random order. Write function for searching whether particular student attended training program or not using linear search and sentinel search.

- b) Write C++ program to store roll numbers of student array who attended training program in sorted order. Write function for searching whether particular student attended training program or not using binary search and Fibonacci search.

Linear Search Performance

- We analyze the successful and unsuccessful searches separately.
- We count how many times the search value is compared against the array elements.
- Successful Search
  - Best Case – 1 comparison
  - Worst Case – N comparisons (N – array size)
- Unsuccessful Search
  - Best Case = Worst Case – N comparisons

Binary Search

- If the array is sorted, then we can apply the binary search technique.
- The basic idea is straightforward. First search the value in the middle position.
- If X is less than this value, then search the middle of the left half next.
- If X is greater than this value, then search the middle of the right half next.

Binary Search :Scenario

We have a sorted array

We want to determine if a particular element is in the array

- **Once found, print or return (index, boolean, etc.)**
- **If not found, indicate the element is not in the collection**

7	12	42		59	71	86	104	212

Binary Search Algorithm

look at "middle" element

if no match then

look *left* (if need smaller) or

right (if need larger)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	7	9	12	33	42	59	76	81	84	91	92	93	99

Look for 42

The Algorithm

```
look at "middle" element
if no match then
  look left or right
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	7	9	12	33	42	59	76	81	[illegible]	[illegible]	92	93	99

Look for 42

The Algorithm

```
look at "middle" element

if no match then

  look left or right
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	7	9	12	33	42	59	76	81	84	91	92	93	99

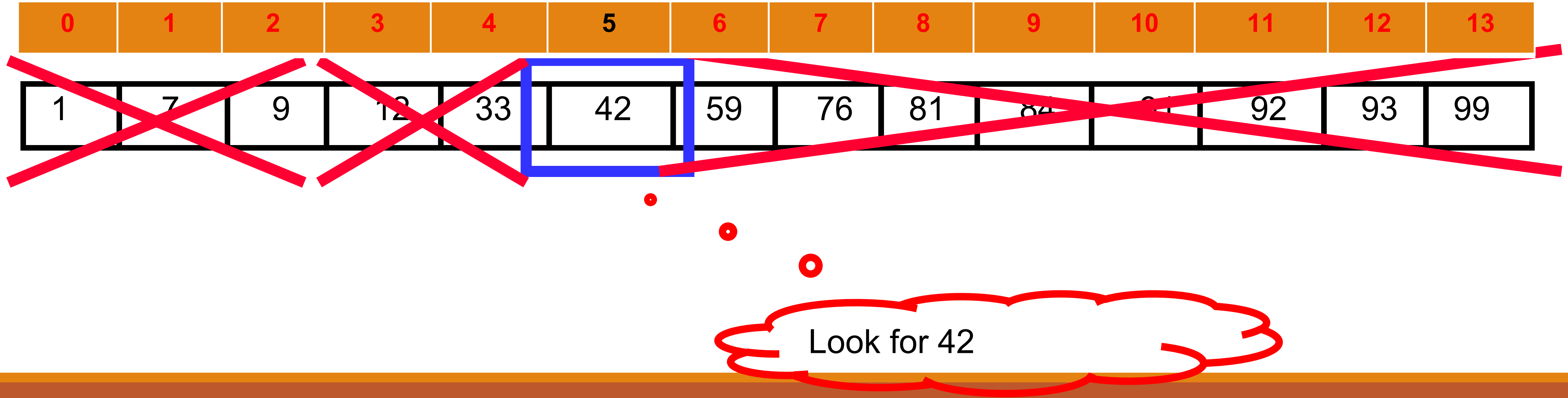
Look for 42

The Algorithm

```
look at "middle" element

if no match then

  look left or right
```



The Binary Search Algorithm

Return found or not found (true or false), so it should be a function.

When move *left* or *right*, change the array boundaries

- **We'll need a first and last**

Time Complexity of Binary Search

The maximum no of elements after 1 comparison $=n/2$

The maximum number of elements after 2 comparisons $=n/2^2$

The maximum number of elements after h comparisons $=n/2^h$

For the lowest value of h elements 1 left

$n/2^h=1$ or $2^h=n$ $h=\log_2(n)=O(\log_2 n)$

Binary Search Algorithm

```
Algorithm binary_search(a[], low, high, key)
{
  while(low<=high)   {
     mid=(low+high)/2;
     if(a[mid]==key)     {
        flag=1;
         return flag; }     //end if
         else if (key<a[mid])
              high=mid-1;
          else
           low=mid+1;
   }   //end while
```

```
If (flag==0)
{   return flag;}

 }//end binary_search
```

Look for 42

1	7	9	12	33	42	59	76	81	84	91	92	93	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Binary search(recursive)

```
Algorithm  binary_search(a[], low, high, key)
{
  if(low<=high)  {
        mid=(low+high)/2;
        if(a[mid]==key)
              return mid;
         else if (key<a[mid])
              return binary_search(a,low,mid-1,key);
         else
            return binary_search(a,mid+1,high,key);}
  return -1;
}
```

Fibonacci Search

- Fibonacci search changes the binary search algorithm slightly
- Instead of halving the index for a search, a Fibonacci number is subtracted from it
- The Fibonacci number to be subtracted decreases as the size of the list decreases
- Note that Fibonacci search sorts a list in a non decreasing order
- Fibonacci search starts searching the target by comparing it with the element at *Fk*th location

Continued…

- Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`
  where each number is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(number) = F(number-1)+ F(number-2);`

The different cases for the search are as follows:

Case 1: if equal the search terminates;

Case 2: if the target is greater and F_1 is 1, then the search terminates with an unsuccessful search;

else the search continues at the right of list with new values of low, high, and mid as

$\texttt{mid} = \texttt{mid} + F_2$, $F_1 = F_{k-4}$ and $F_2 = F_{k-5}$

Case 3: if the target is smaller and F_2 is 0, then the search terminates with unsuccessful search;

else the search continues at the left of list with new values of low, high, and mid as

$\texttt{mid} = \texttt{mid} - F_2$, $F_1 = F_{k-3}$ and $F_2 = F_{k-4}$

The search continues by either searching at the left of mid or at the right of mid in the list.

- $F_0 = 0$
- $F_1 = 1$
- $F_2 = 1$
- $F_3 = 2$
- $F_4 = 3$
- $F_5 = 5$
- $F_6 = 8$
- $F_7 = 13$
- $F_8 = 21$
- $F_9 = 34$
- $F_{10} = 55$

A=

0	1	2	3	4	5	6	7	8	9
6	14	23	36	55	67	76	78	81	89

Key =78

N=10

Compute F_k such that F_k>=10

Fib(7)=13 >10 hence k=7

Compute initial values of mid

Mid=n-F_{k-2}+ 1 $F_1=F_{k-2}$ $F_2=F_{k-3}$

The target to be searched is compared with A[mid]

$F_{1=}$ fibo(7-2)=fibo(5)=5 $F_{2=}$ fibo(7-3)=fibo(4)=3

Mid=10-F_{k-2}+1=10-5+1=6 (76)

1. 78>A[6-1] (If f1!=1) mid=mid+F_2 =6+3=9
   F_1= F_1 – F_2= 5-3 so, F_1 =2
   $\mathbf{F_2}$= F_2 - F1= 3 – 2 = **1**

2. 78<a[9-1] mid=mid- F_2 =9-1=**8**
   t= F_1 – F_2= 2 – 1 =1
   F_1= F_2 so, F_1 = F_2 $\mathbf{F_{1=}}$ **1**
   F_2 =t F_2=1

3. 78 <a[8] mid=mid-F_2 = 8-1 =7

4. 78==a[8-1]

$A=6,14,23,36,55,67,76,78,81,89$

- $F_0 = 0$
- $F_1 = 1$
- $F_2 = 1$
- $F_3 = 2$
- $F_4 = 3$
- $F_5 = 5$
- $F_6 = 8$
- $F_7 = 13$
- $F_8 = 21$
- $F_9 = 34$
- $F_{10} = 55$

Key =81

N=10

Compute F_k such that $F_k>=10$

Fib(7)=13 >10 hence k=7

Compute initial values of mid

$Mid=n-F_{k-2}+1$

$F_1=F_{k-2}$

$F_2=F_{k-3}$

The target to be searched is compared with A[mid]

$F_{1=}$ fibo(7-2)=fibo(5)=5

$F_{2=}$ fibo(7-3)=fibo(4)=3

$Mid=10-F_{k-2}+1=10-5+1=6$ (76)

1. 81>A[6] mid=mid+ $F_2=6+3=9$

   $F_1=f_{k-4}=F_{7-4}=F_3$ $F_1=2$

   $F_2=F_{k-5}=F_{2} =1$ $F_2=1$

2. 81<a[9] mid=mid-F2=9-1=8

   $F_1=f_{k-3}=F_{7-3}=F_4$ $F_1=3$

   $F_2=F_{k-4}=F_{7-4} =F3$ $F_2=2$

3. 81=a[8]

```
Algorithm fib_search(a,n)
{
    find fk >=n;
    initially f1= f(k-2) ; f2=f(k-3);
    mid=n-f(k-2)+1
    while key != a[mid-1]
    {
    if (mid<0 or key>a[mid-1])
        {
        if f1==1  return -1;
            mid=mid+f2;
            f1=f1-f2
            f2=f2-f1
        }
    else
        {
            if f2==0
              return -1;
            mid = mid - f2
            t = f1 - f2
            f1 = f2
            f2 = t
        }
    }
    return mid
}
```

Time Complexity of Fibonacci Search

- **Fibonacci search is more efficient than binary search for large- sized lists**

- **However, it is inefficient in case of small lists**

- **The number of comparisons is of the order of *n, and the time complexity is O(log(n))***

Sorting

General Sort Concepts

Sort Order :

- Data can be ordered either in ascending order or in descending order
- The order in which the data is organized, either ascending order or descending order, is called sort order

Sort Stability

- **A sorting method** is said to be stable if at the end of the method, identical elements occur in the same relative order as in the original unsorted set

- Sort Efficiency
- Sort efficiency is a measure of the relative efficiency of a sort

Continued…

Passes

- During the sorted process, the data is traversed many times
- Each traversal of the data is referred to as a sort pass
- In addition, the characteristic of a sort pass is the placement of one or more elements in a sorted list

Sorting

Sorting takes an unordered collection and makes it an ordered one.

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
5	12	35	42	77	101

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.

- An ***internal sort*** requires that the collection of data fit entirely in the computer's main memory.

- We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.

In Place Sort

- The amount of extra space required to sort the data is constant with the input size.

Stable sort algorithms

- A stable sort keeps equal elements in the same order
- This may matter when you are sorting data according to some characteristic

  Example: sorting students by test scores

Ann	98		Ann	98
Bob	90		Joe	98
Dan	75		Bob	90
Joe	98		Sam	90
Pat	86		Pat	86
Sam	90		Zöe	86
Zöe	86		Dan	75
original array			stably sorted	

Unstable sort algorithms

- An unstable sort may or may not keep equal elements in the same order
- Stability is usually not important, but sometimes it is important

original array		unstably sorted	
Ann	98	Joe	98
Bob	90	Ann	98
Dan	75	Bob	90
Joe	98	Sam	90
Pat	86	Zöe	86
Sam	90	Pat	86
Zöe	86	Dan	75

Types of Sorting Algorithms

There are many, many different types of sorting algorithms, but the primary ones are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Radix Sort

Bubble sort("Bubbling Up" the Largest Element)

Traverse a collection of elements

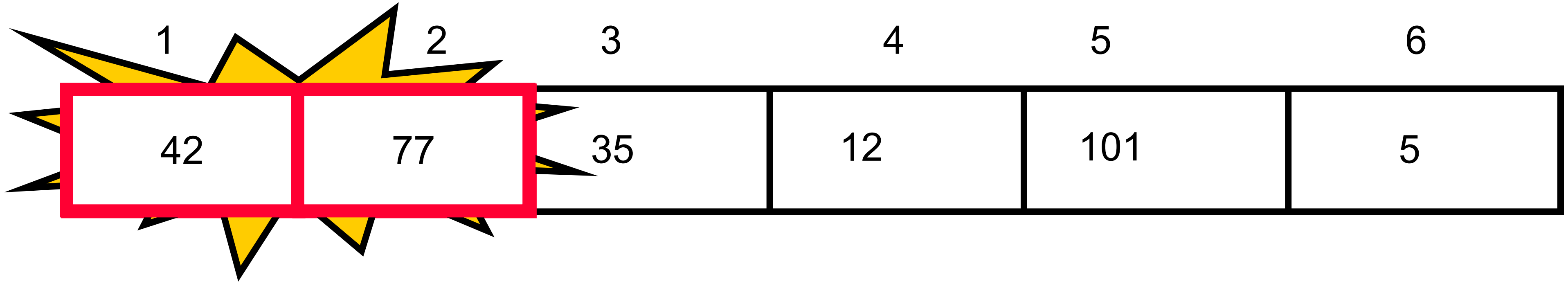
- **Move from the front to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
77	42	35	12	101	5

"Bubbling Up" the Largest Element

Traverse a collection of elements

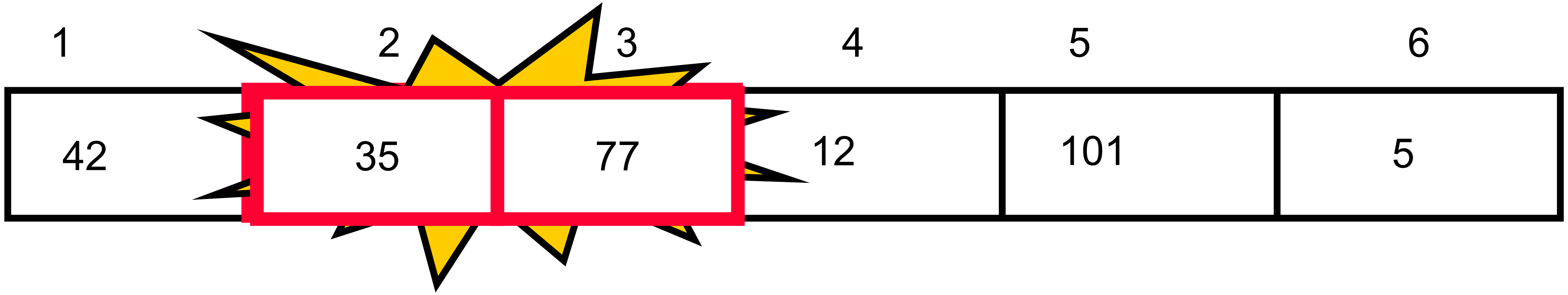
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

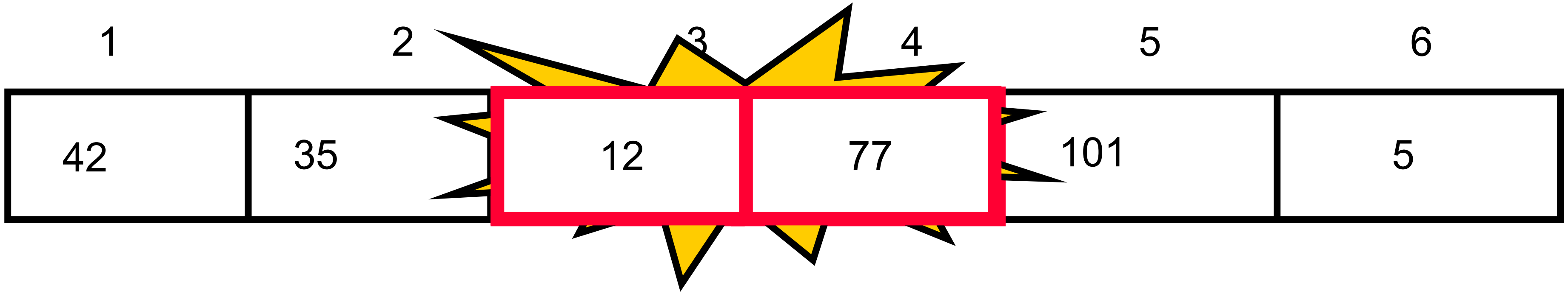
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the 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 Up" the Largest Element

Traverse a collection of elements

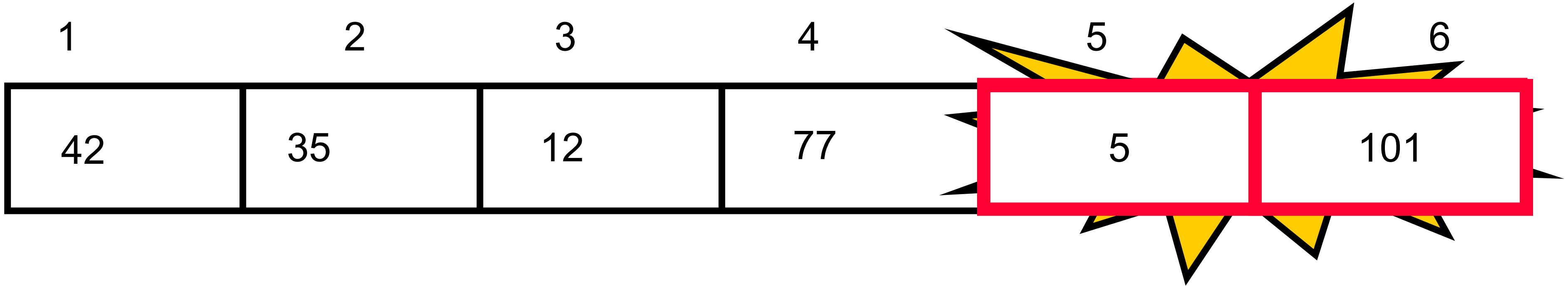
- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling Up" the 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 Up" the 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**

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

“Bubbling” All the Elements

N - 1

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

1	2	3	4	5	6
5	12	35	42	77	101

Reducing the Number of Comparisons

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

Bubble sort

```
Algorithm  bubble()
{
        for  i =0 to n-1
         {
          for   j=0   to n-i-1
          {
                 if  a[j]>a[j+1]
                 {
                   swap(a[j],a[j+1])
                 }
          }
        }
      display(a,n);
}
```

Analysis of Bubble Sort

The time complexity for each of the cases is given by the following:

Average-case complexity = O(n^2)

Best-case complexity = O(n^2)

Worst-case complexity = O(n^2)

Selection Sort

Idea:

- Find the smallest element in the array
- Exchange it with the element in the first position
- Find the second smallest element and exchange it with the element in the second position
- Continue until the array is sorted

Disadvantage:

- Running time depends only slightly on the amount of order in the file

Selection Sort

```
void selectionsort()
{
    for i = 0 to n-2 {
     minpos = i;
        for  j = i+1 to n-1  {
            if a[j] < a[minpos]  {
        minpos = j; }
    }
        if (minpos!=i ){
            temp = a[i];
        a[i] = a[minpos];
        a[minpos] = temp;
    }
}
```

8	4	6	9	2	3	1

Example

8	4	6	9	2	3	1
1	4	6	9	2	3	8
1	2	6	9	4	3	8
1	2	3	9	4	6	8
1	2	3	4	9	6	8
1	2	3	4	6	9	8
1	2	3	4	6	8	9
1	2	3	4	6	8	9

Analysis of Selection Sort

Best Case: $O(n^2)$

Worst Case: $O(n^2)$

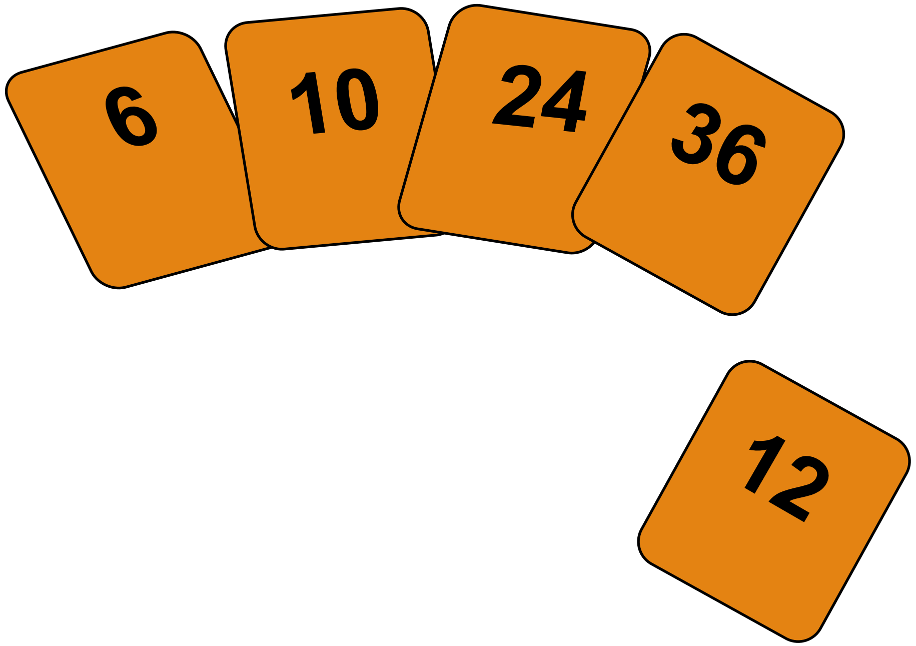
Average case: $O(n^2)$

Insertion Sort

Idea: like sorting a hand of playing cards

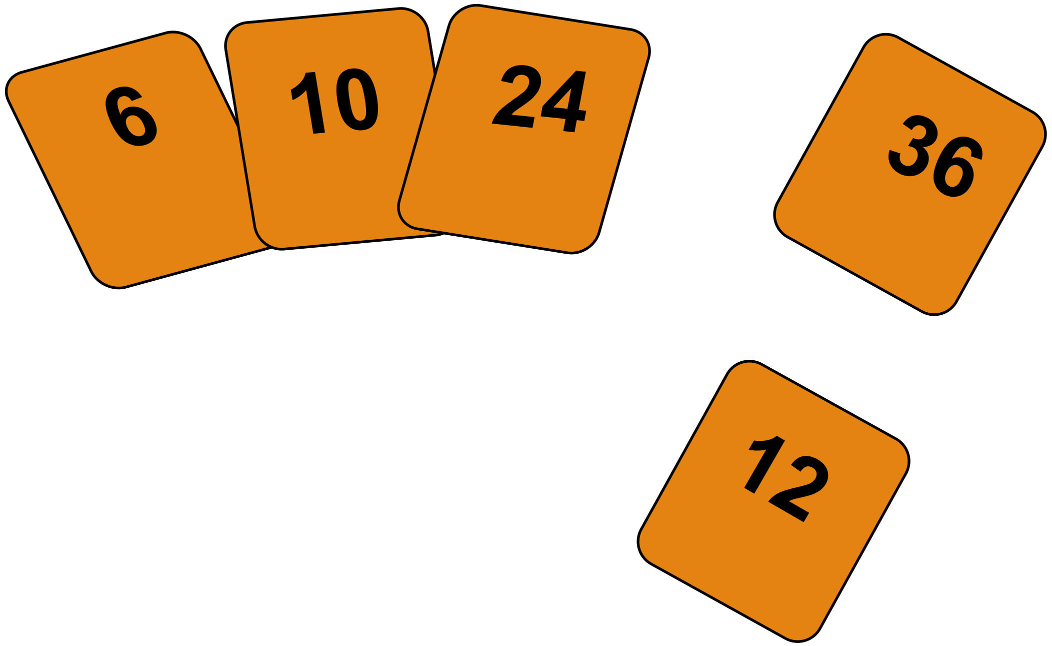
- Start with an empty left hand and the cards facing down on the table.
- Remove one card at a time from the table, and insert it into the correct position in the left hand
  - compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted
  - these cards were originally the top cards of the pile on the table

Insertion Sort

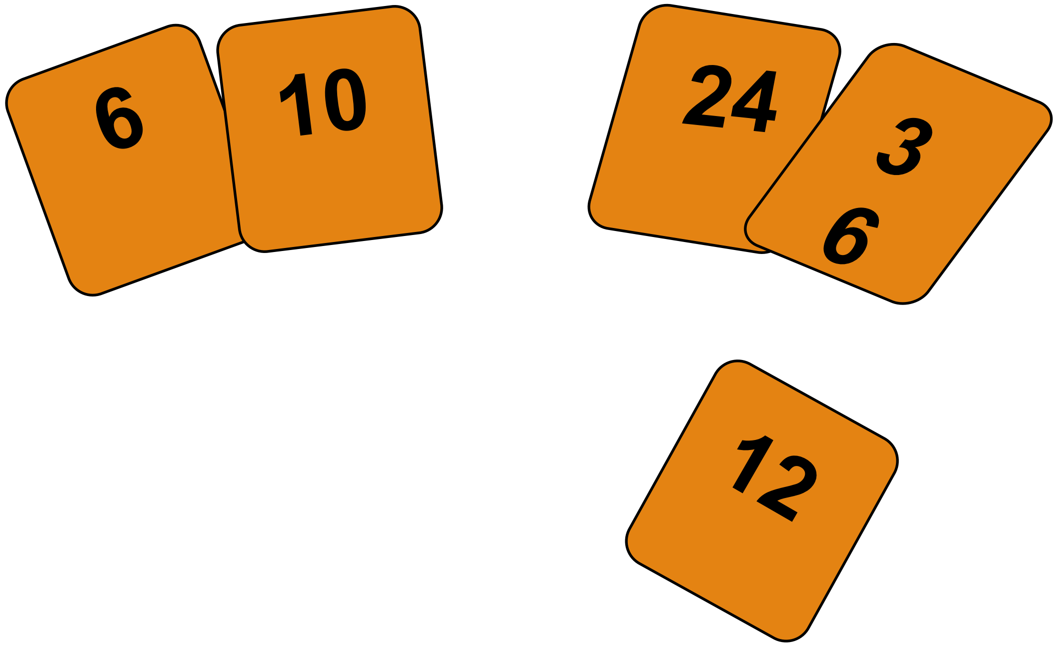


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

input array

5 2 4 6 1 3

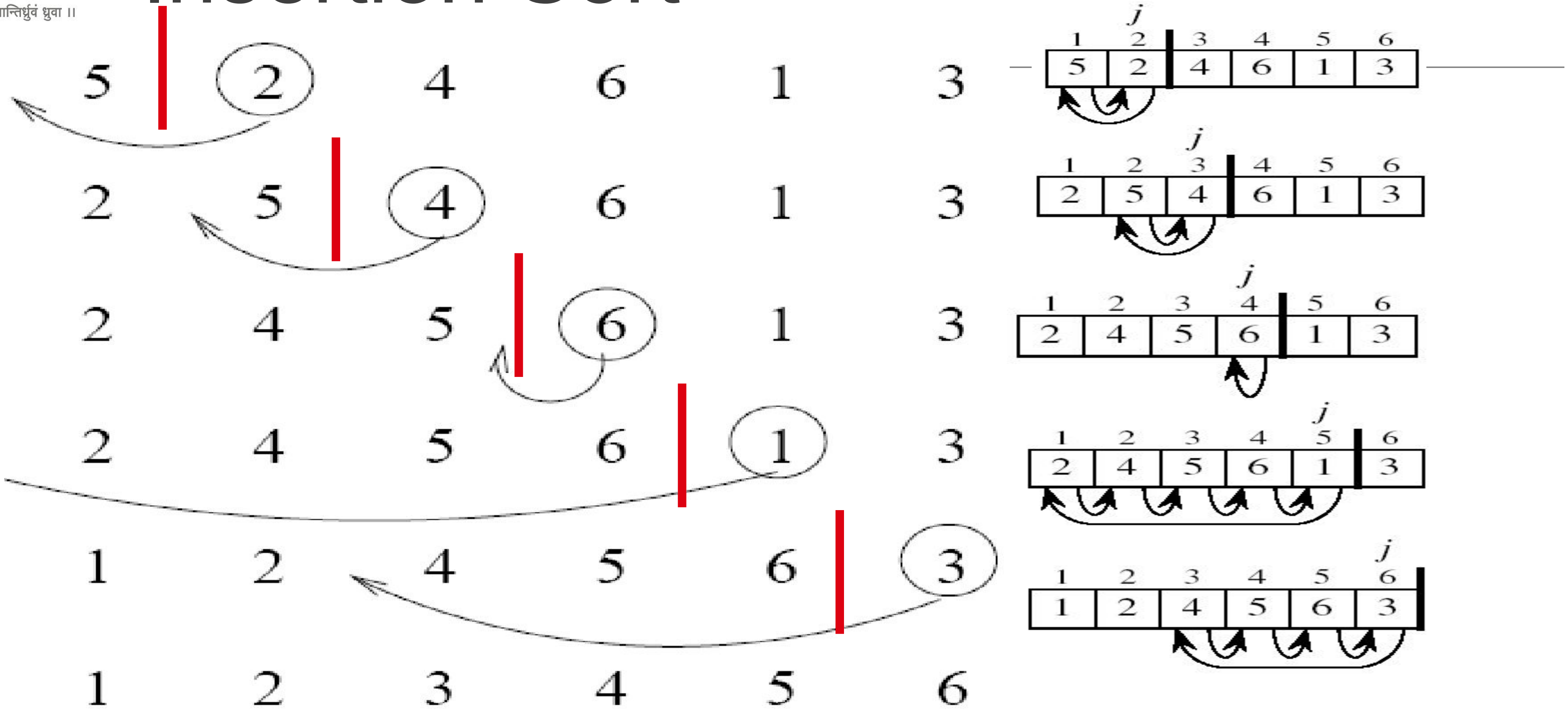
at each iteration, the array is divided in two sub-arrays:

left sub-array right sub-array

2 5 | 4 6 1 3

sorted unsorted

Insertion Sort



Insertion Sort

```
void insertionSort( arr[],  n)
{

  for (i = 1; i < n; i++)
  {
     key = arr[i];
     j = i-1;

/* Move elements of arr[0..i-1], that are greater than
key, to one position ahead of their current position */
       while (j >= 0 && arr[j] > key)
       {
          arr[j+1] = arr[j];
          j = j-1;
       }
       arr[j+1] = key;
   }
}
```

Analysis of Insertion Sort

If the data is initially sorted, only one comparison is made on each pass so that the sort time complexity is O(n)

The number of interchanges needed in both the methods is on the average $(n^2)/4$, and in the worst cases is about $(n^2)/2$

Shell sort

```
void shell_sort(int A[],int n]
{
    gap=n/2;
    do
    {
        do
        {
            swapped=0;
            for(i = 0; i < n- gap ; i++)
                if(A[ i ] > A[ i + gap ])
                {
                    swap();
                    swapped=1;
                }
        } while(swapped == 1);
    }while((gap=gap/2) >= 1);
}
```

https://www.w3resource.com/ODSA/AV/Sorting/shellsortAV.html

Shell sort- Complexity Analysis

1. Complexity in the **Best Case: O(n*Log n)**

   The total number of comparisons for each interval (or increment) is equal to the size of the array when it is already sorted.

1. Complexity in the **Average Case: O(n*log n)**

   It's somewhere around O. (n1.25)

1. Complexity in the **Worst-Case** Scenario: Less Than or Equal to **O (n^2)**

   Shell sort's worst-case complexity is always less than or equal to **O. (n^2)**

The degree of complexity is determined by the interval picked. The above complexity varies depending on the increment sequences used. The best increment sequence has yet to be discovered.

General Concept of Divide & Conquer

Given a function to compute on *n* inputs, the divide-and-conquer strategy consists of:

- splitting the inputs into k distinct subsets, $1<k\leq n$, yielding k subproblems.
- solving these subproblems
- combining the subsolutions into solution of the whole.
- if the subproblems are relatively large, then divide_Conquer is applied again.
- if the subproblems are small, the are solved without splitting.

Control Abstraction for Divide and Conquer

```
Divide_Conquer(problem P)
{
   if Small(P) return S(P);
   else {
     divide P into smaller instances P1, P2, …, Pk, k≥1;
     Apply Divide_Conquer to each of these subproblems;
     return Combine(Divide_Conque(P1), Divide_Conque(P2),…,
   Divide_Conque(Pk));
   }
}
```

Three Steps of The Divide and Conquer Approach

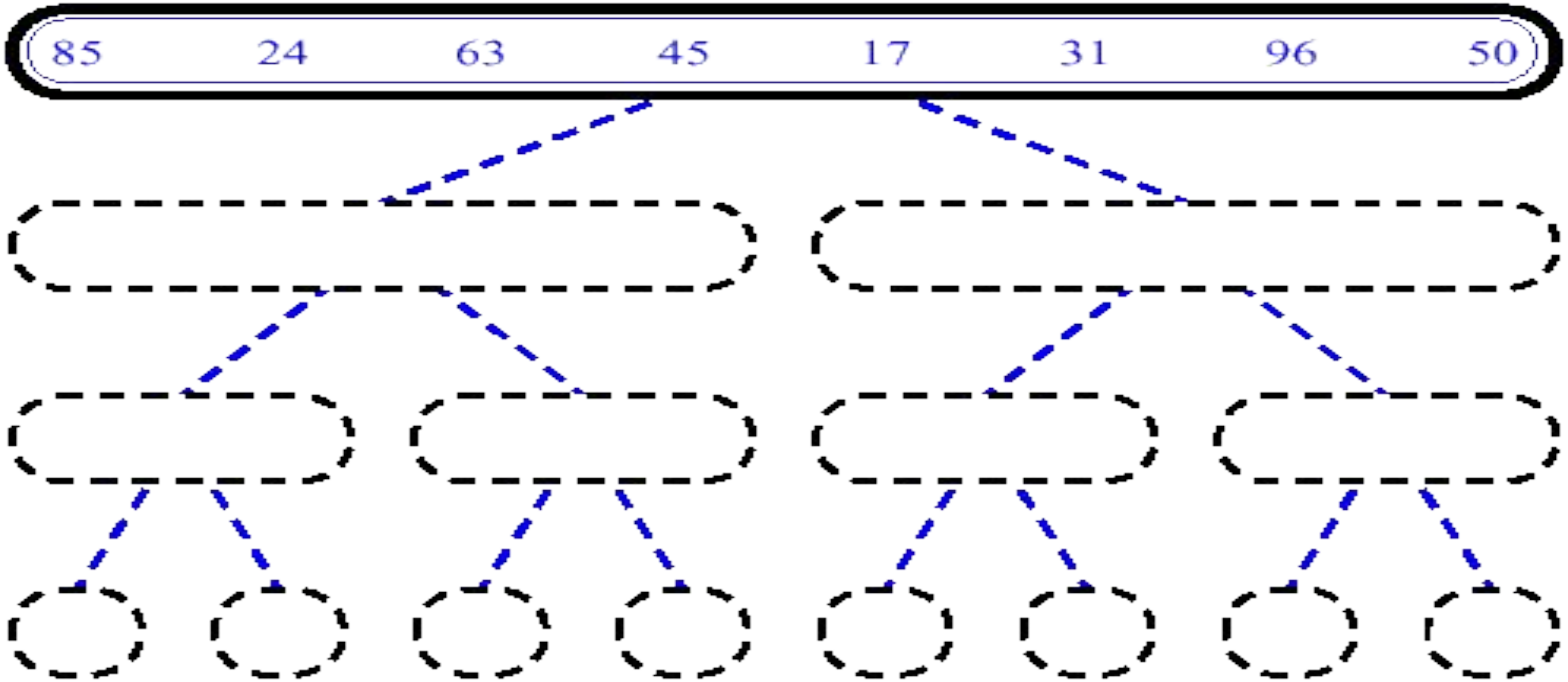
The most well known algorithm design strategy:

1. **Divide** the problem into two or more smaller subproblems.

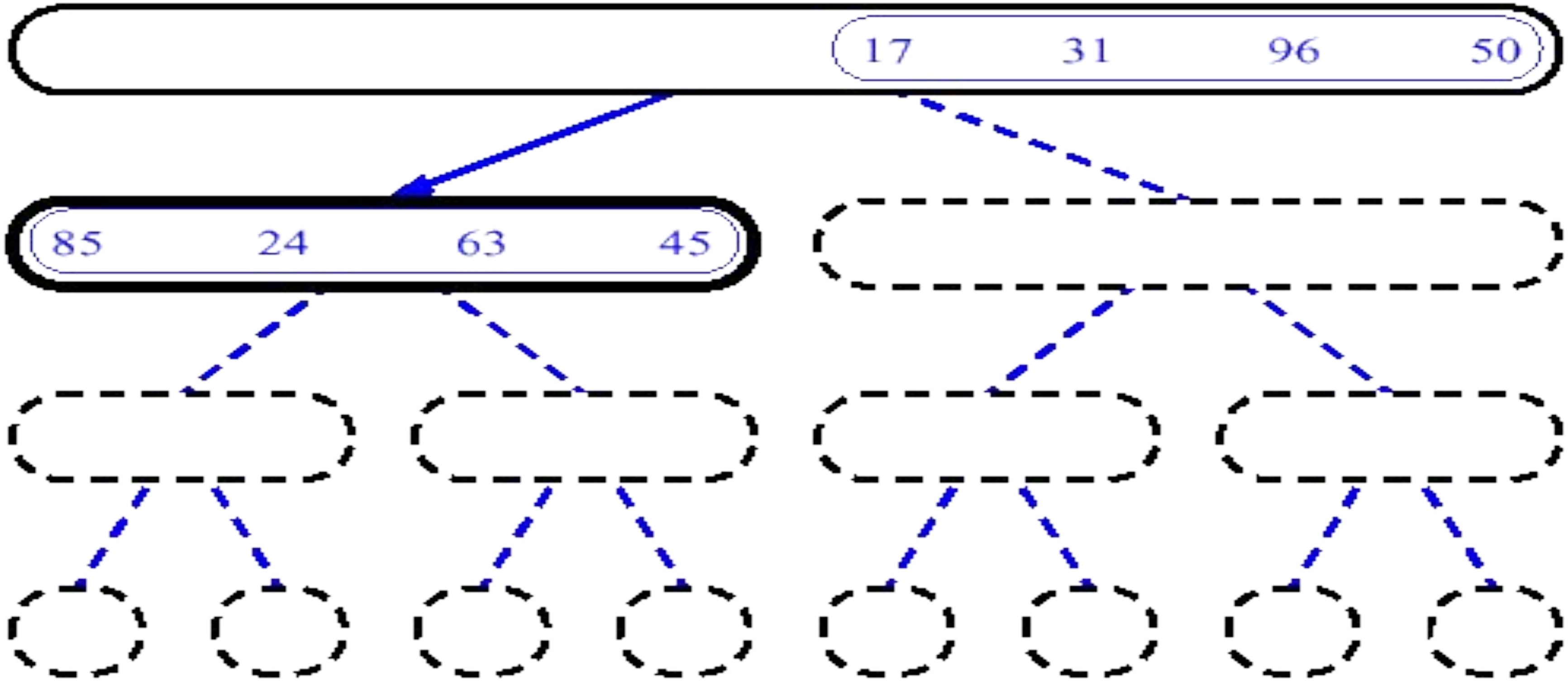
2. **Conquer** the subproblems by solving them recursively.

3. **Combine** the solutions to the subproblems into the solutions for the original problem.

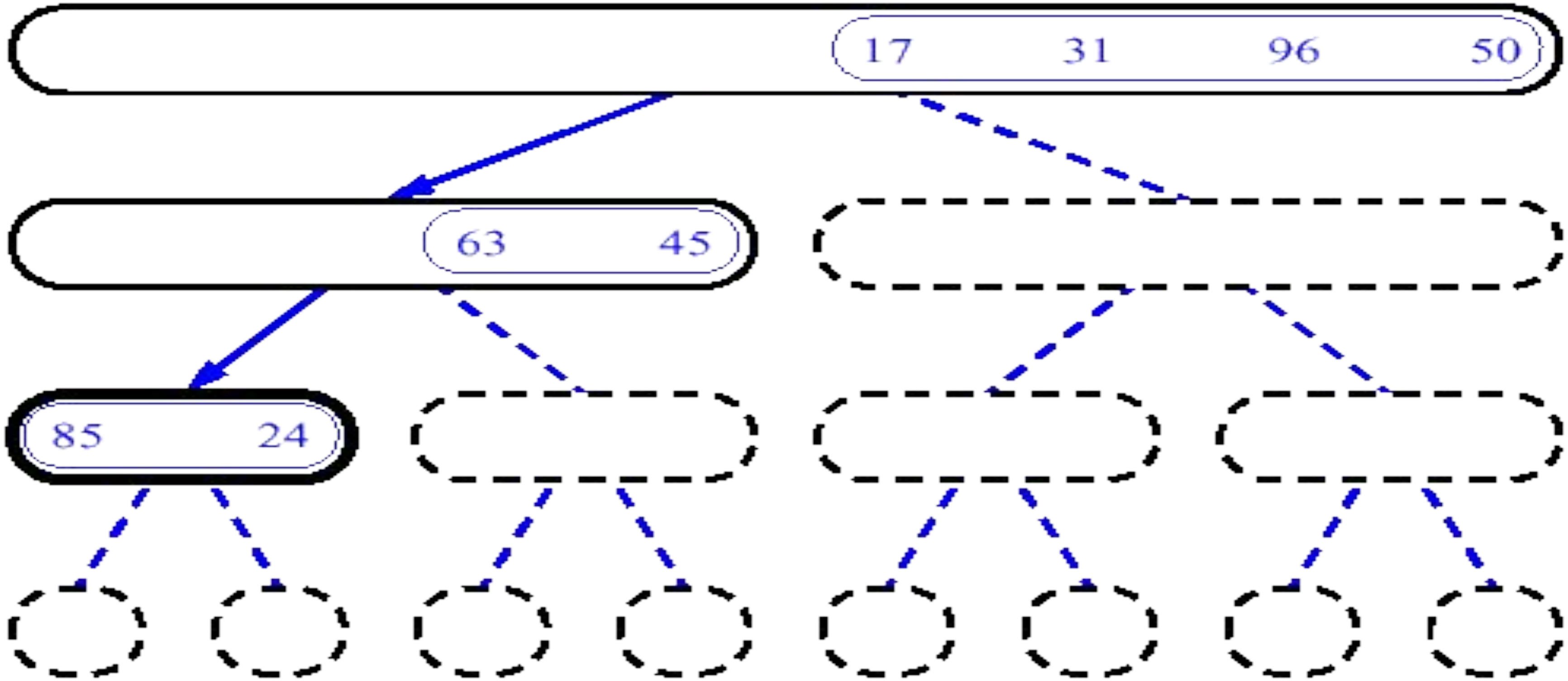
MergeSort (Example) - 1



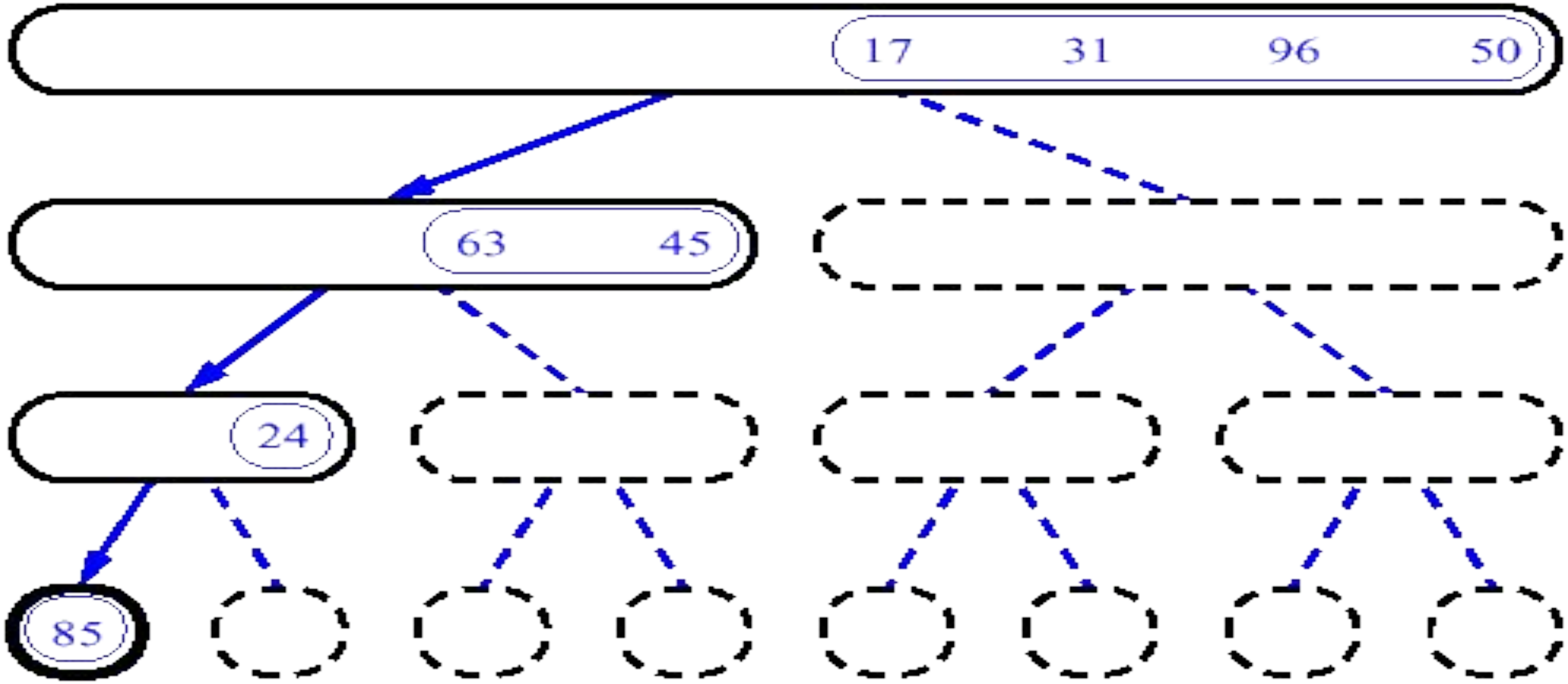
MergeSort (Example) - 2



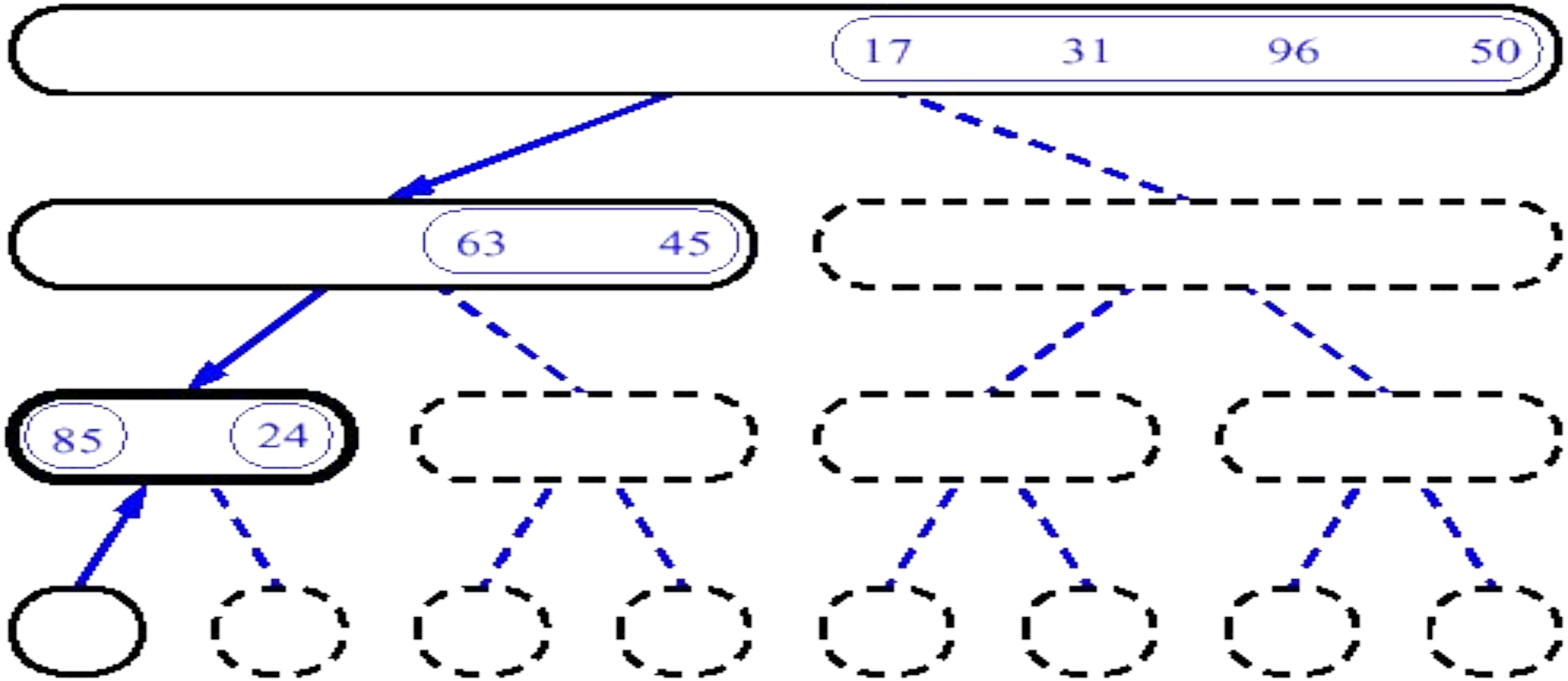
MergeSort (Example) - 3



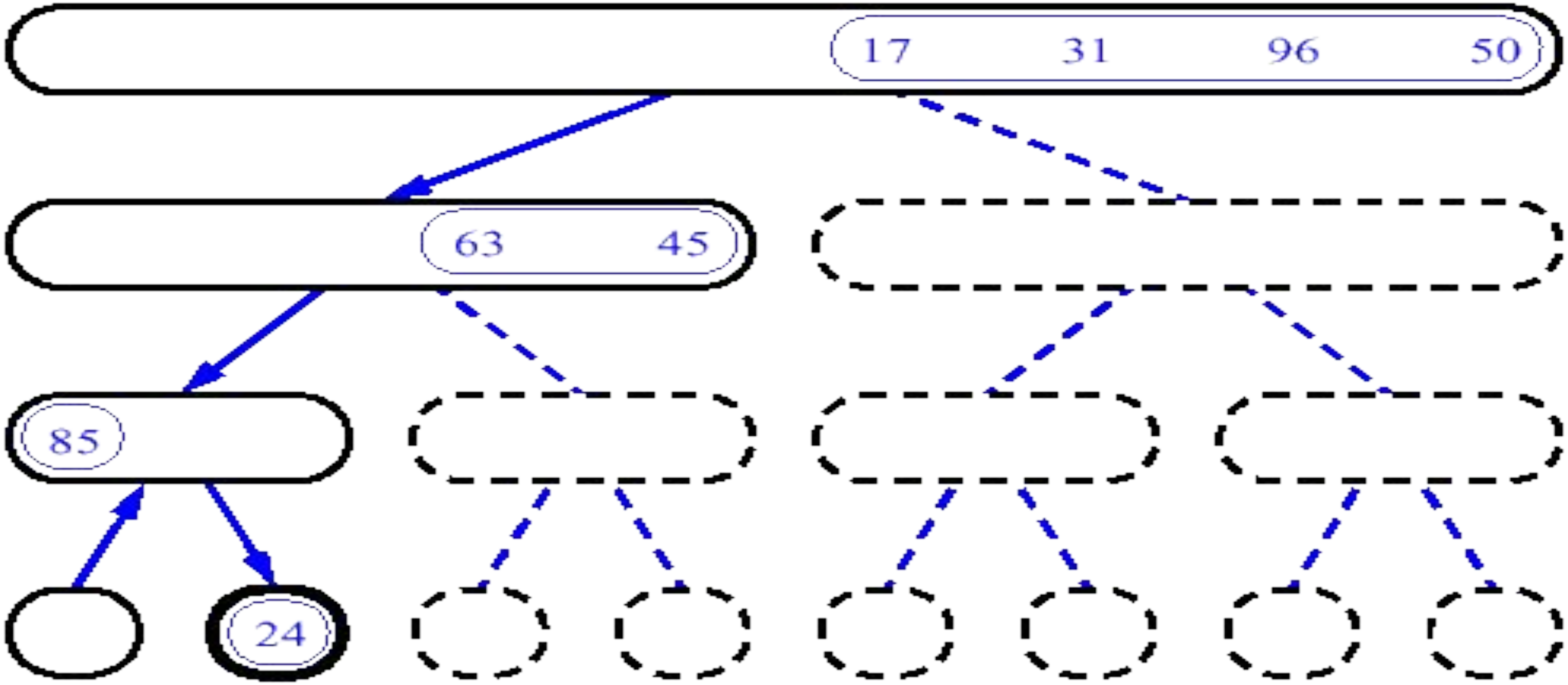
MergeSort (Example) - 4



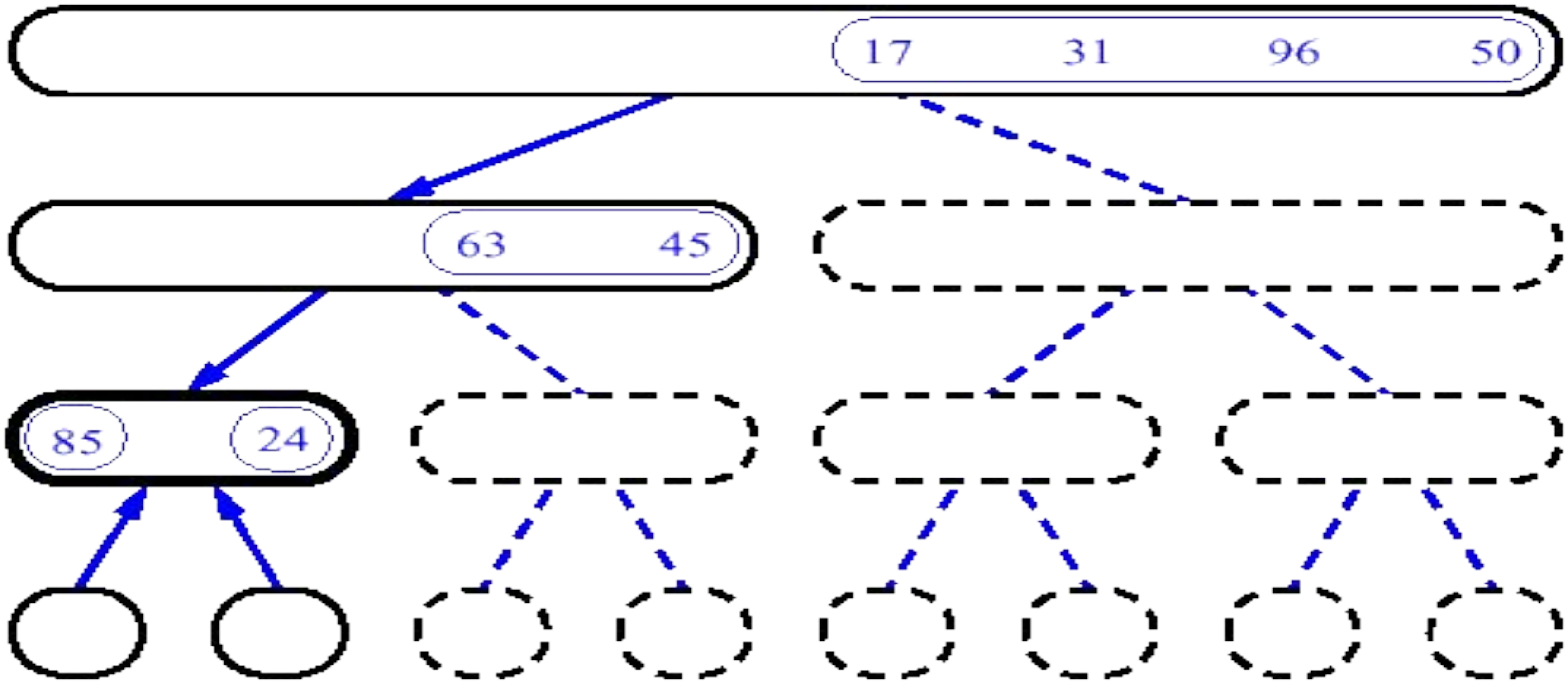
MergeSort (Example) - 5



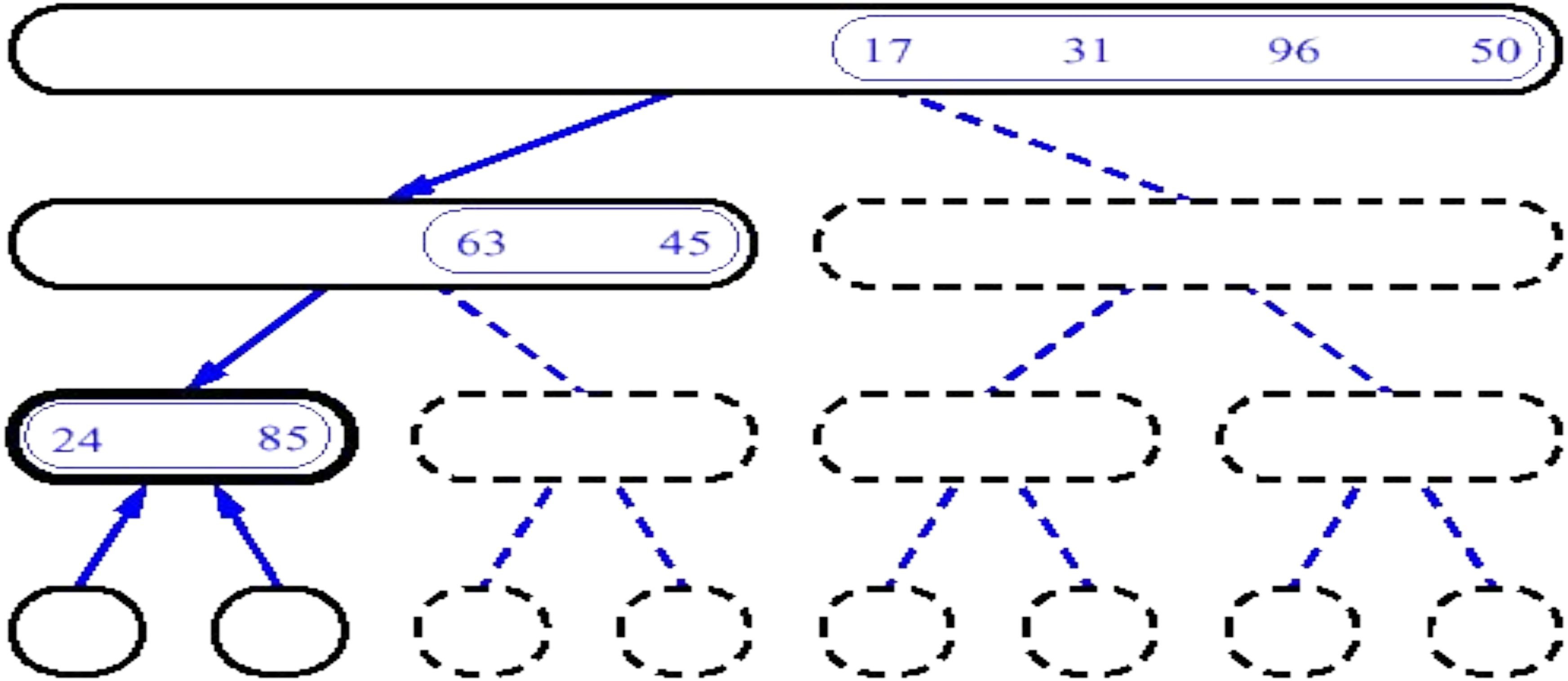
MergeSort (Example) - 6



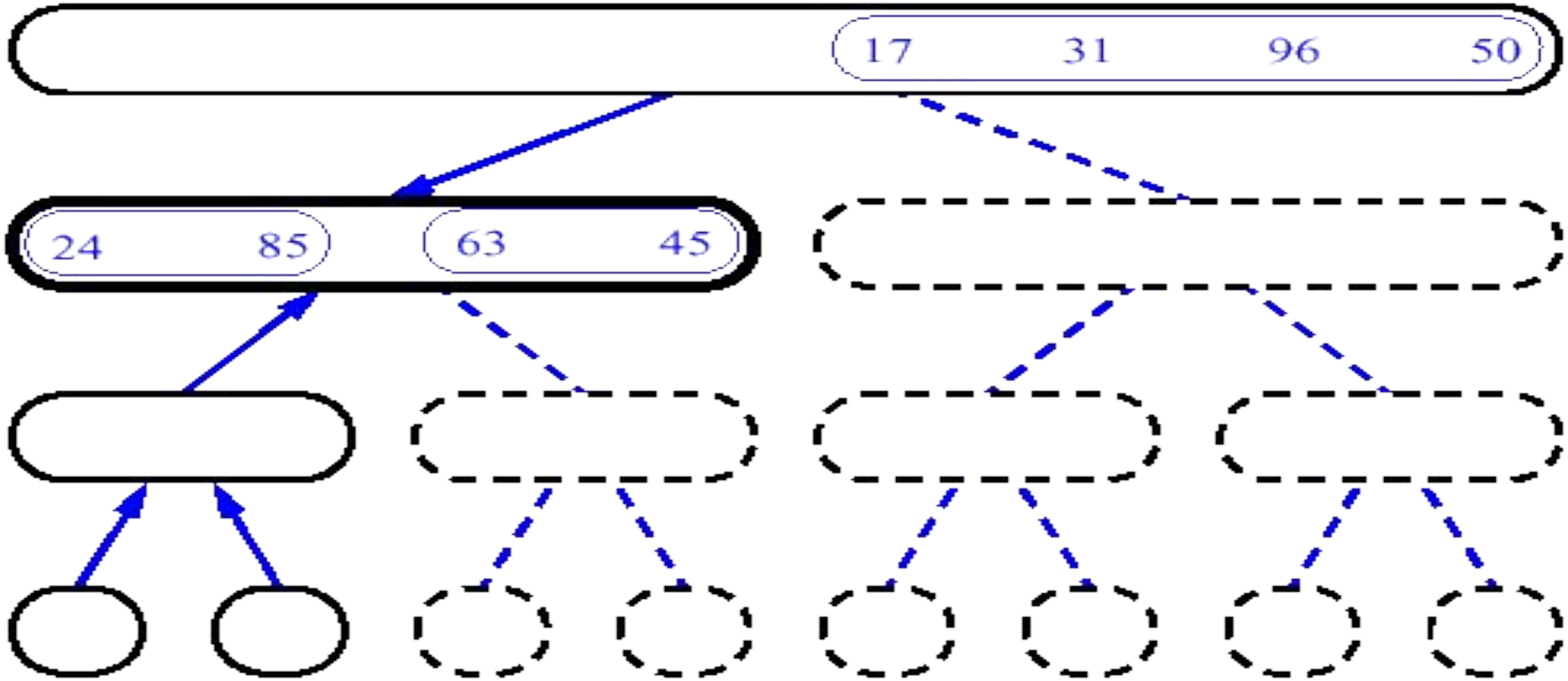
MergeSort (Example) - 7



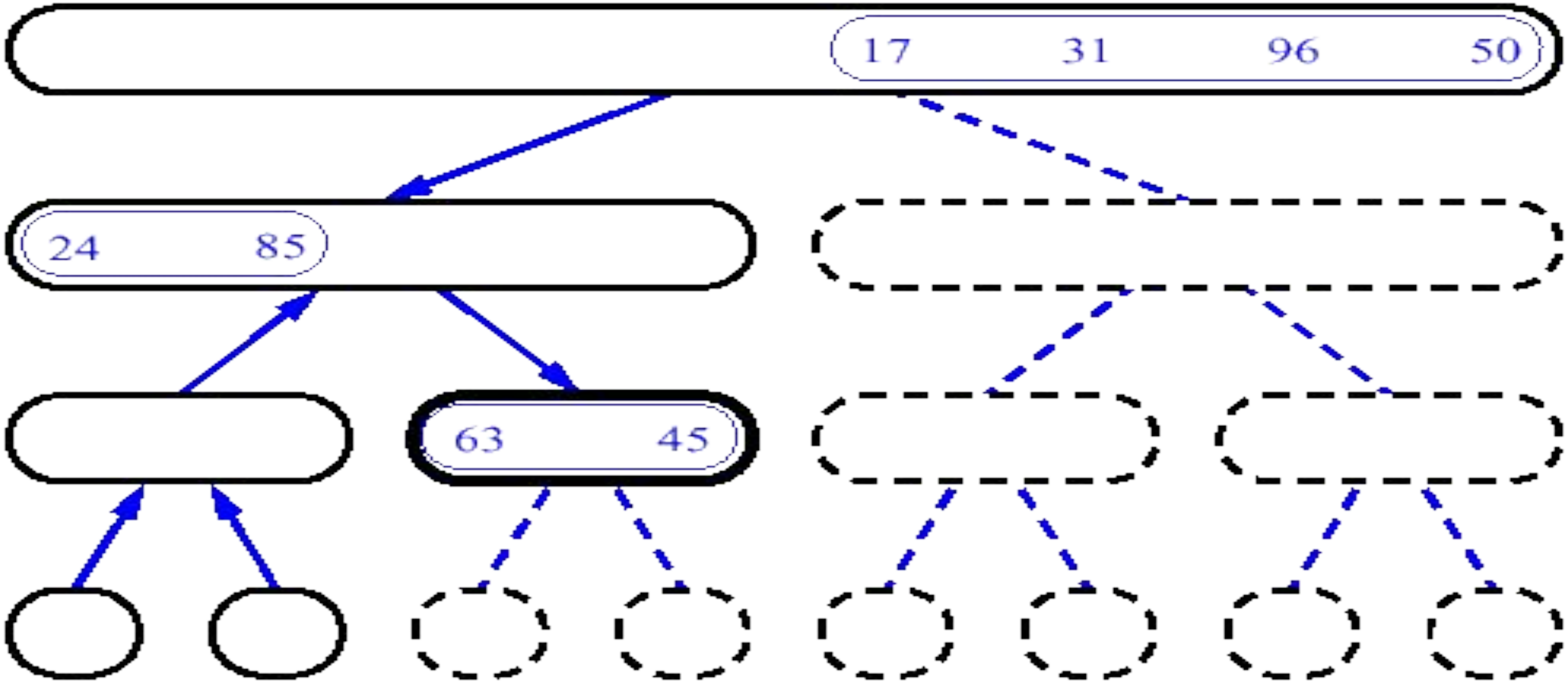
MergeSort (Example) - 8



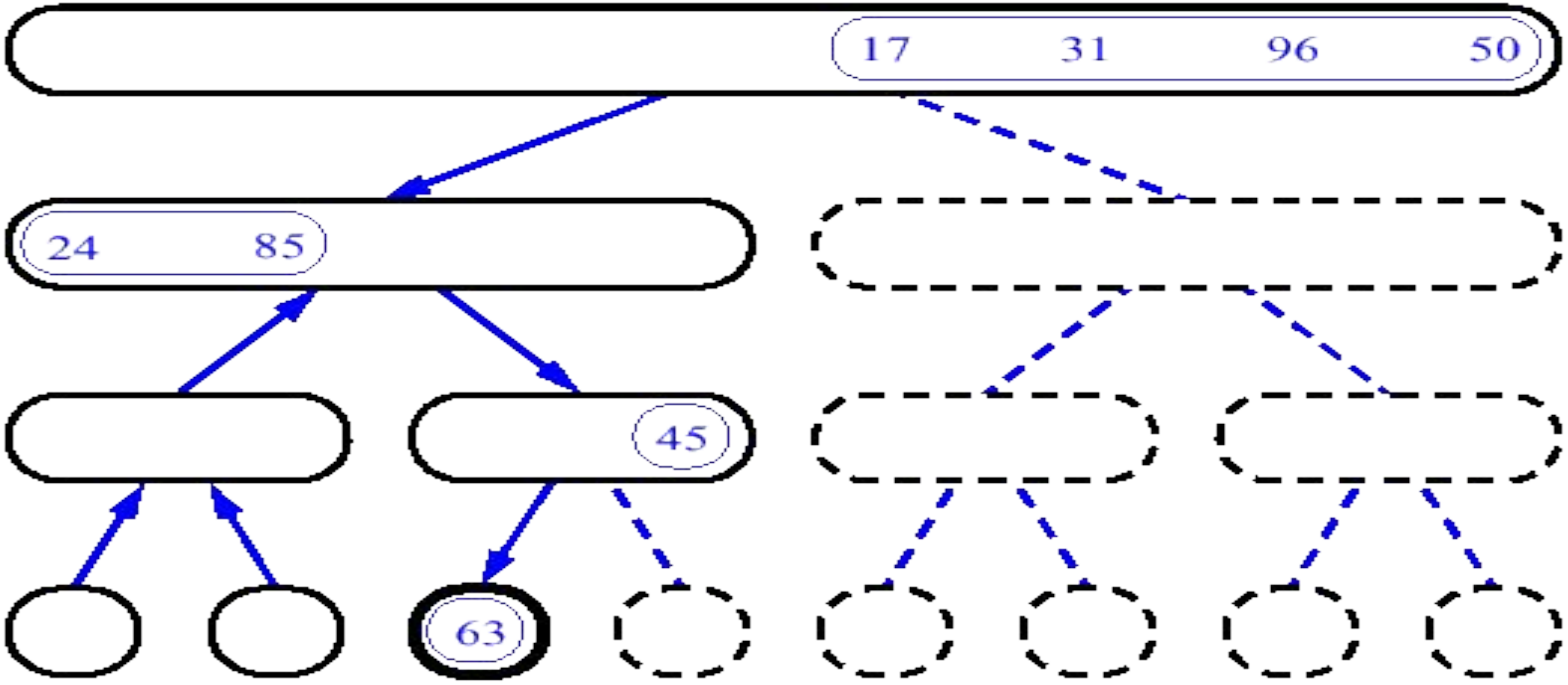
MergeSort (Example) - 9



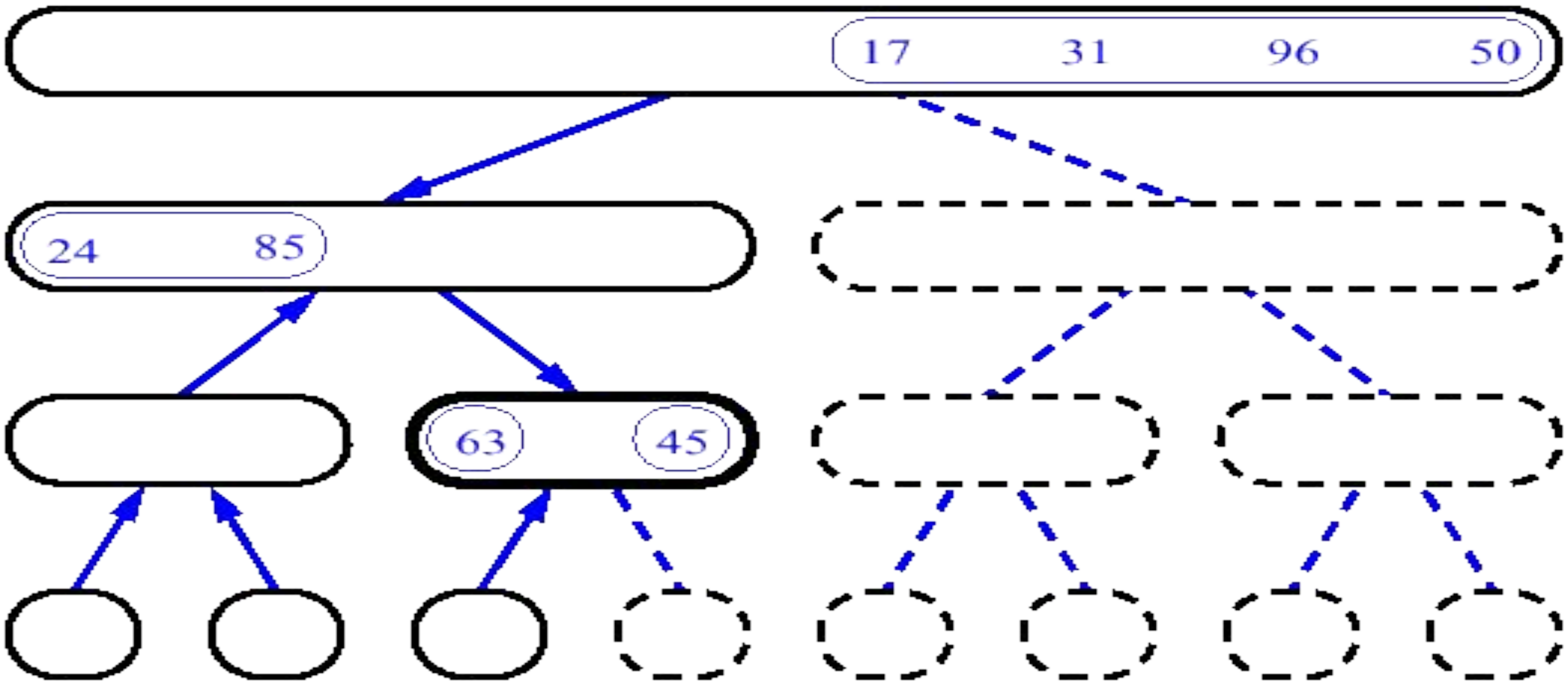
MergeSort (Example) - 10



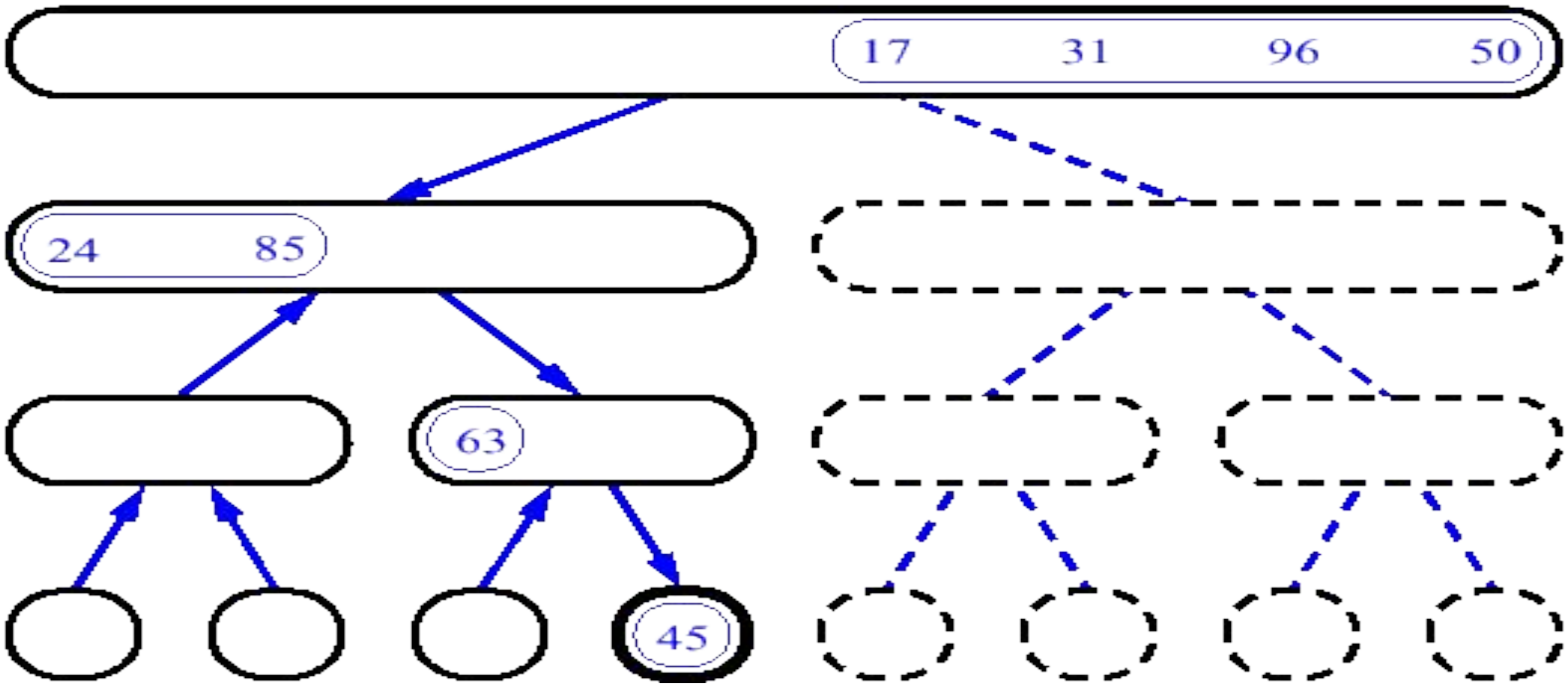
MergeSort (Example) - 11



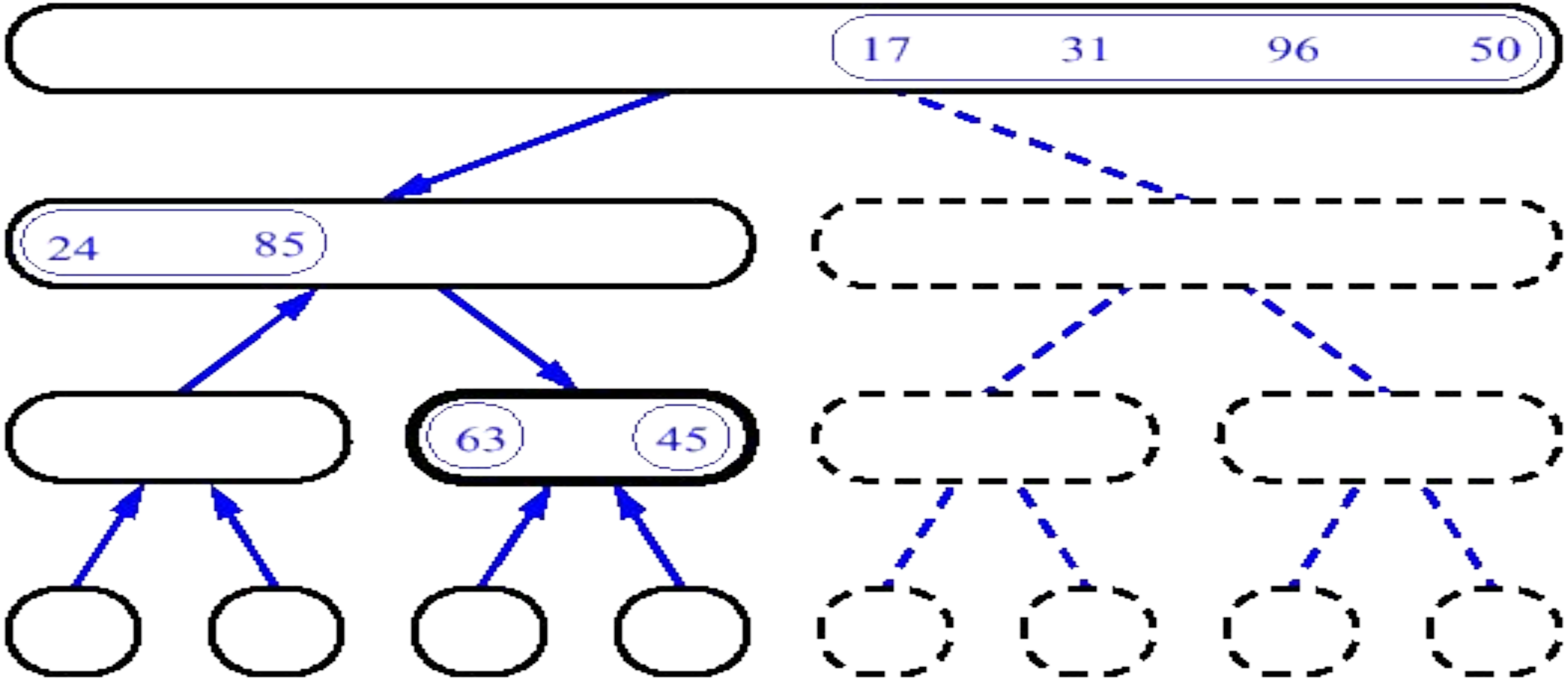
MergeSort (Example) - 12



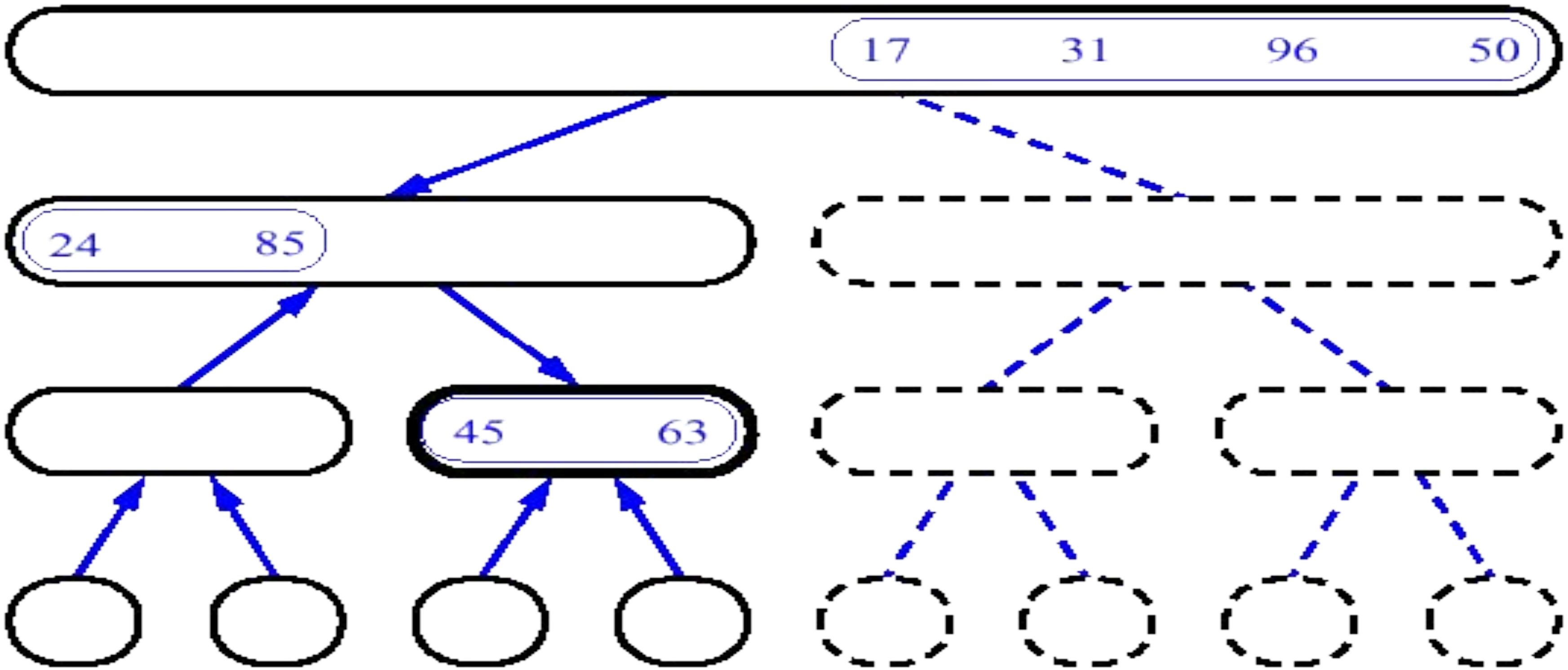
MergeSort (Example) - 13



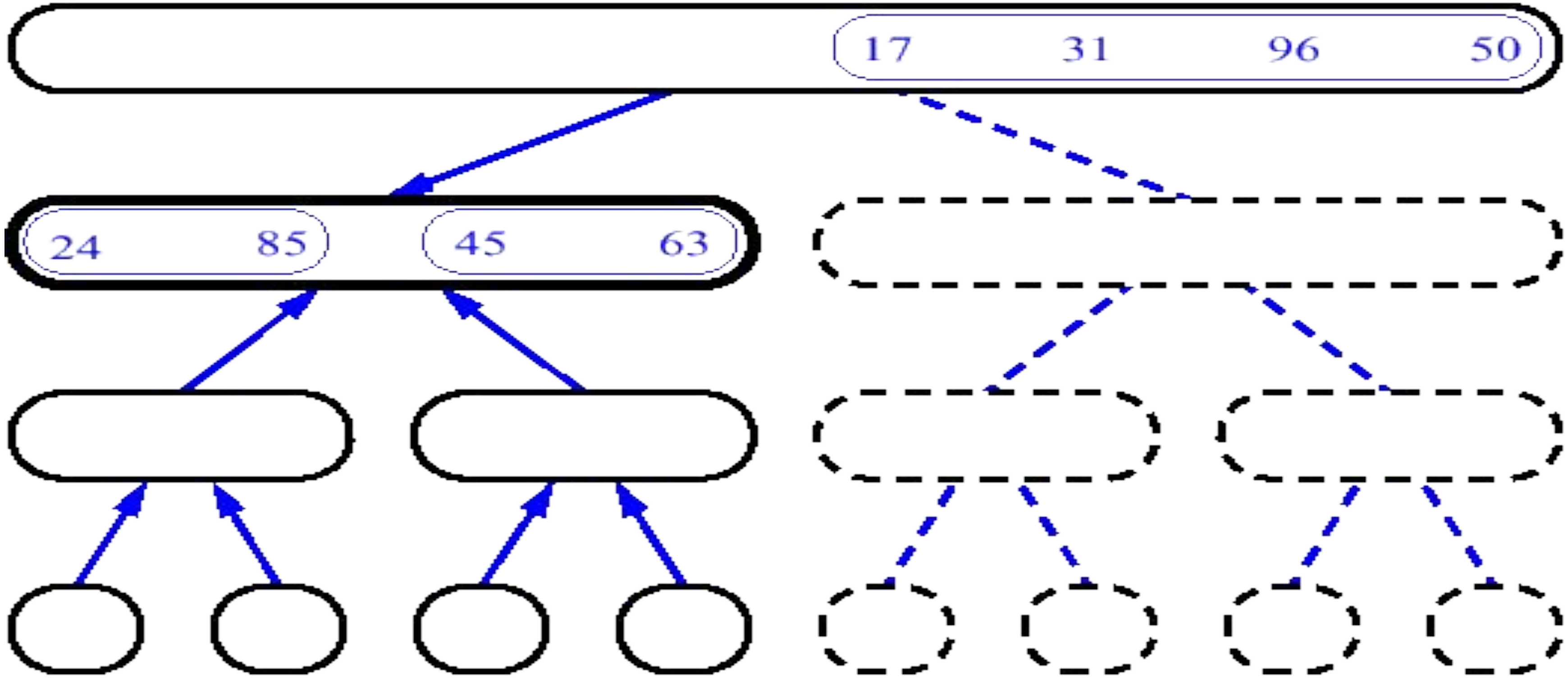
MergeSort (Example) - 14



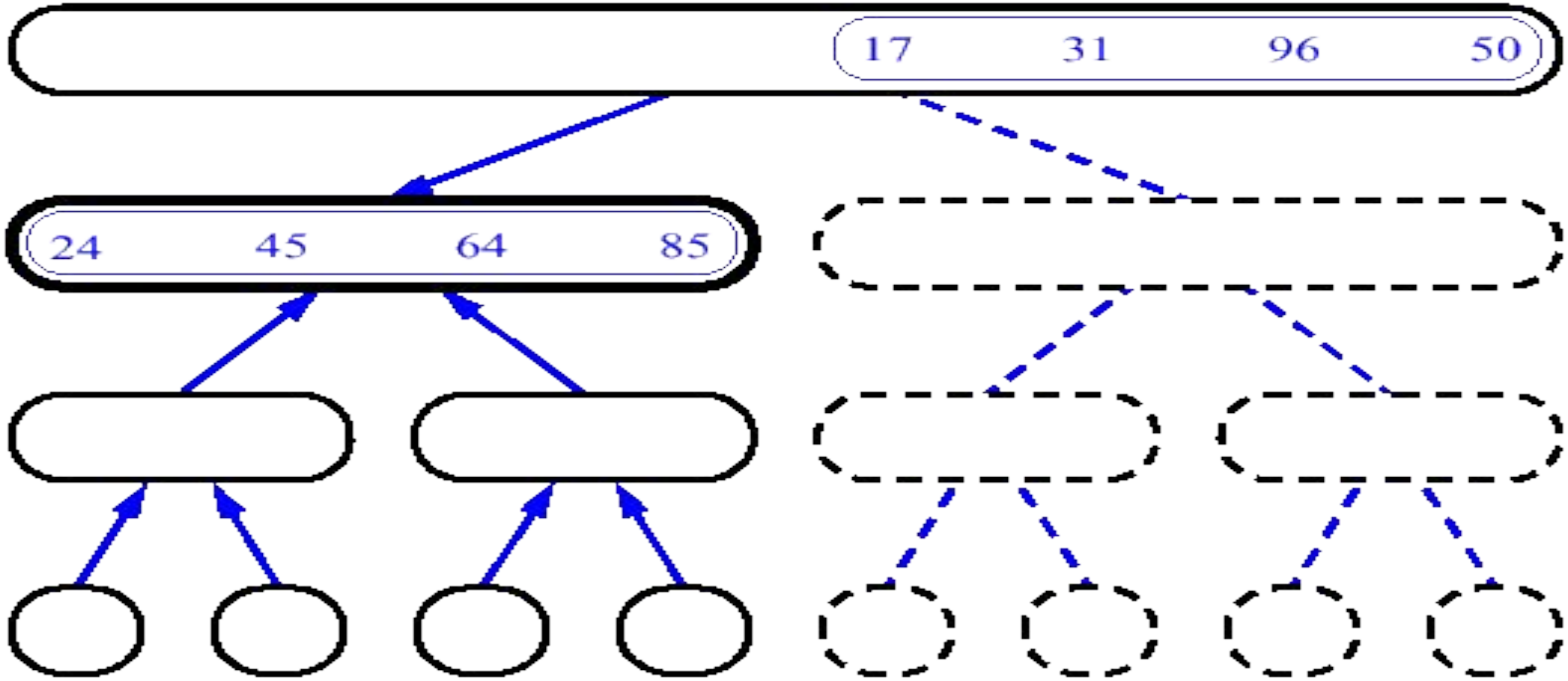
MergeSort (Example) - 15



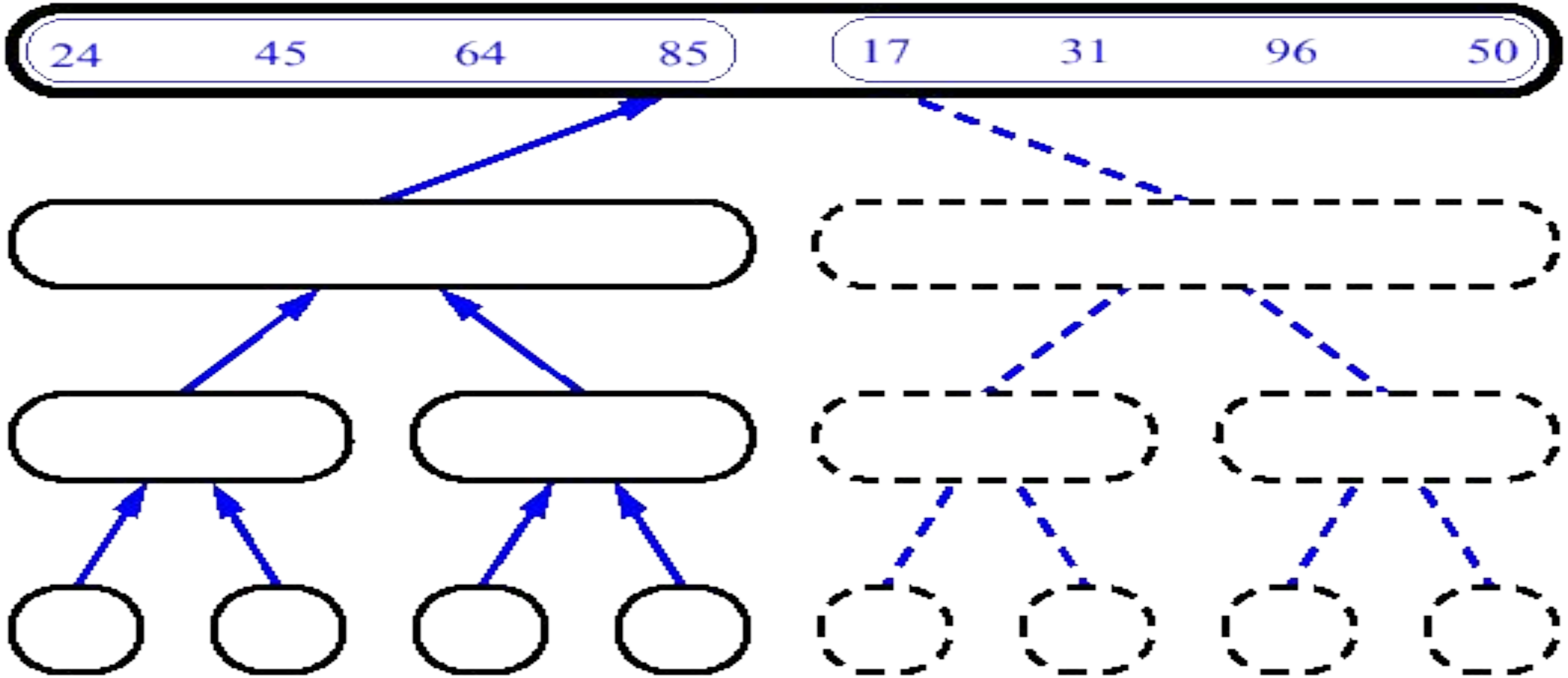
MergeSort (Example) - 16



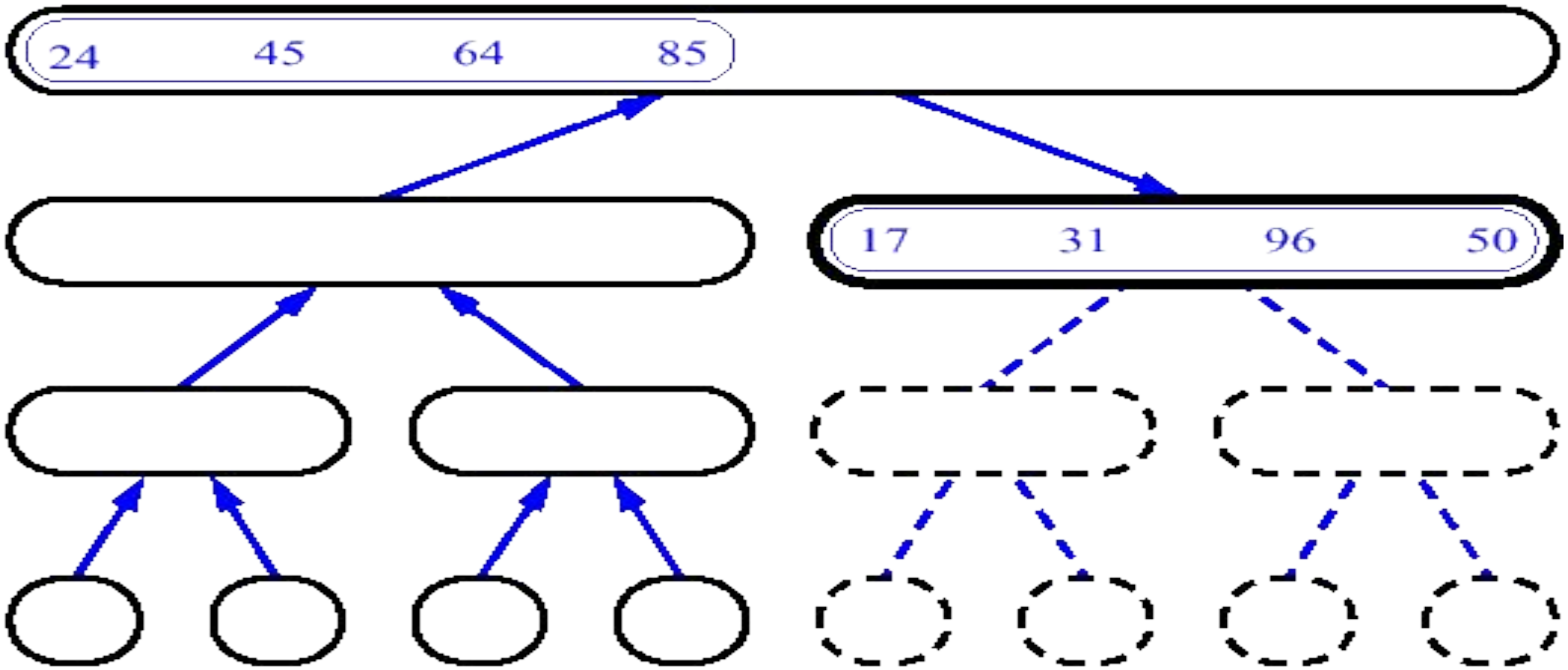
MergeSort (Example) - 17



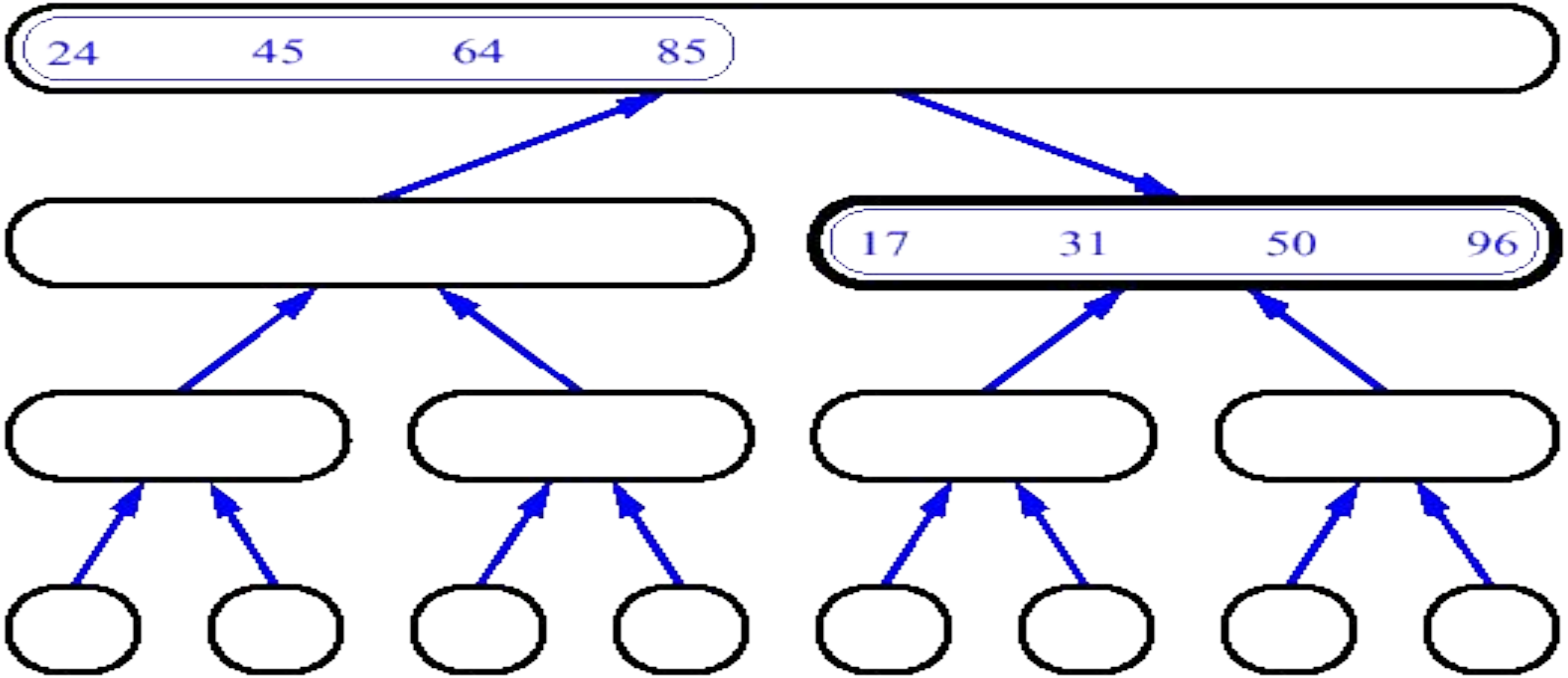
MergeSort (Example) - 18



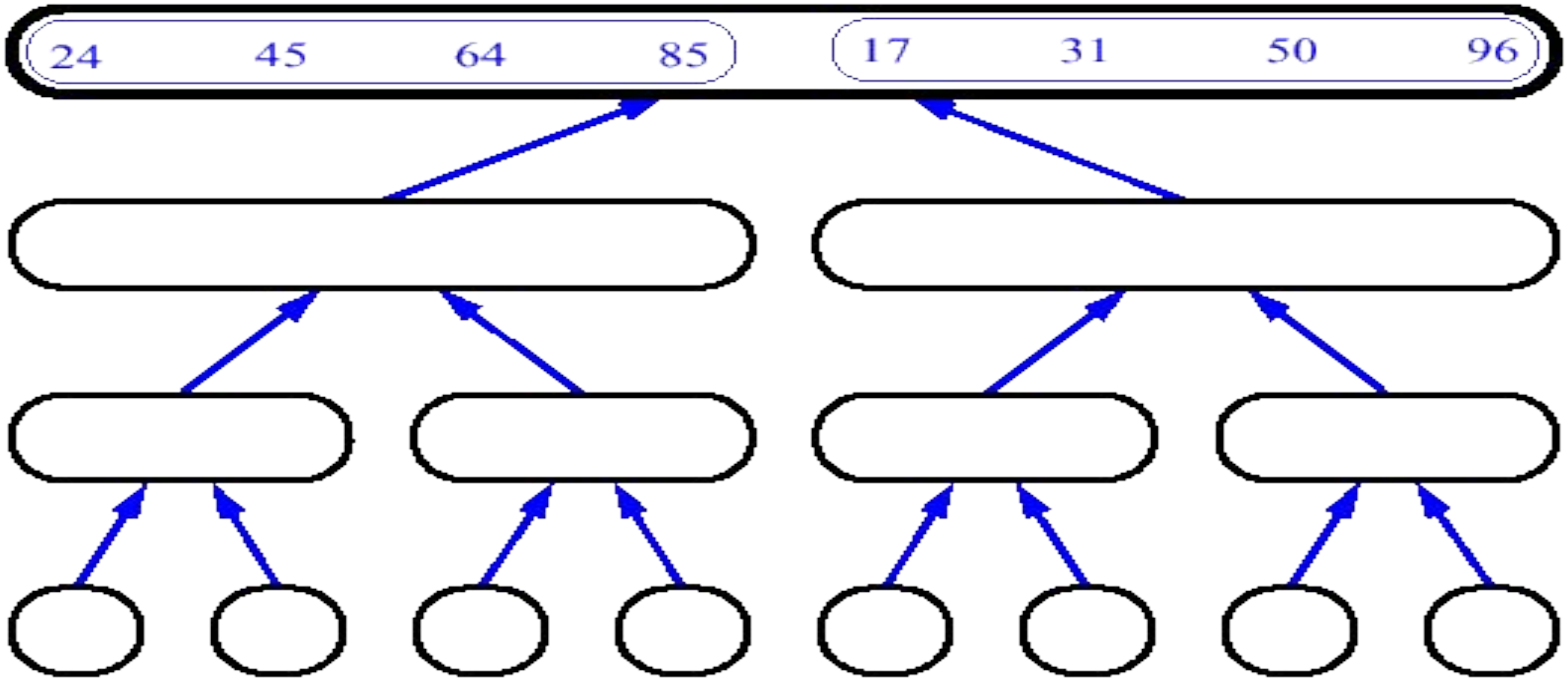
MergeSort (Example) - 19



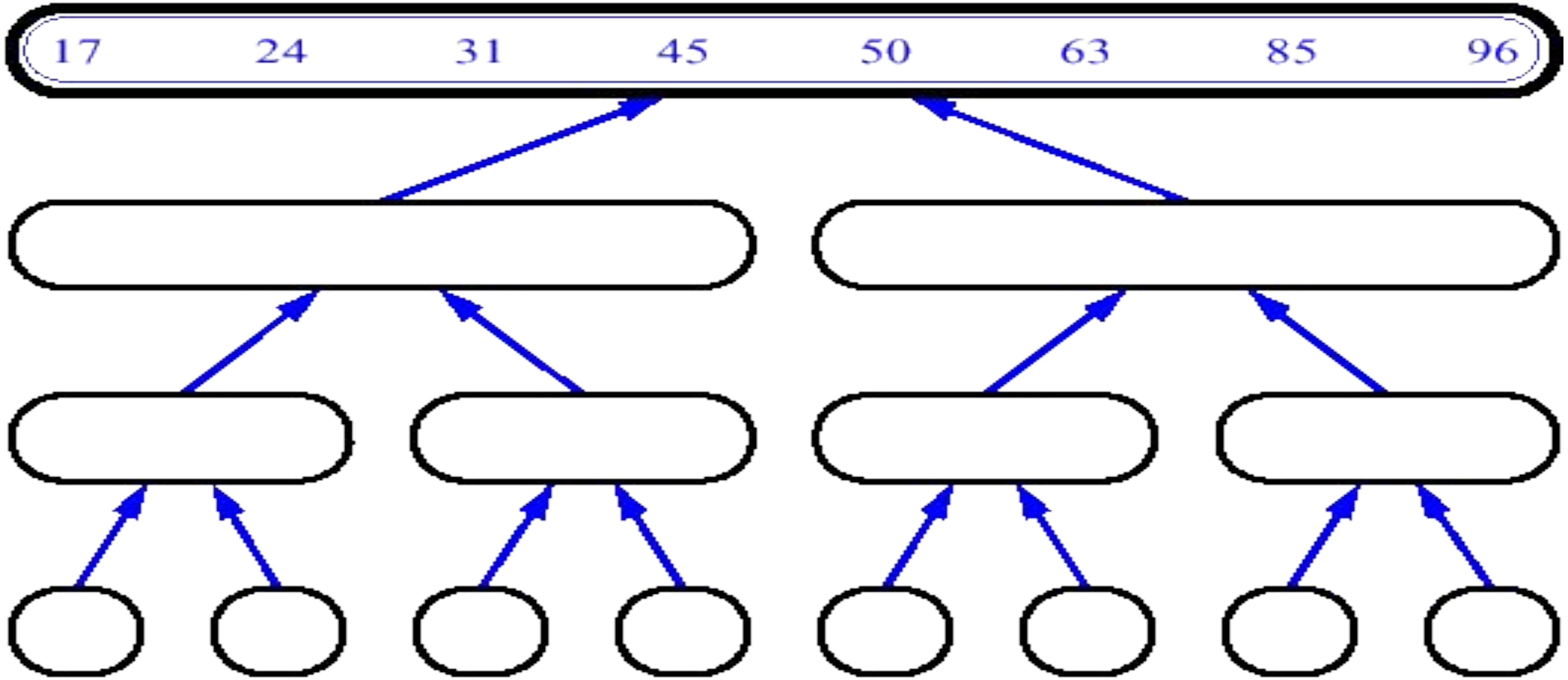
MergeSort (Example) - 20



MergeSort (Example) - 21

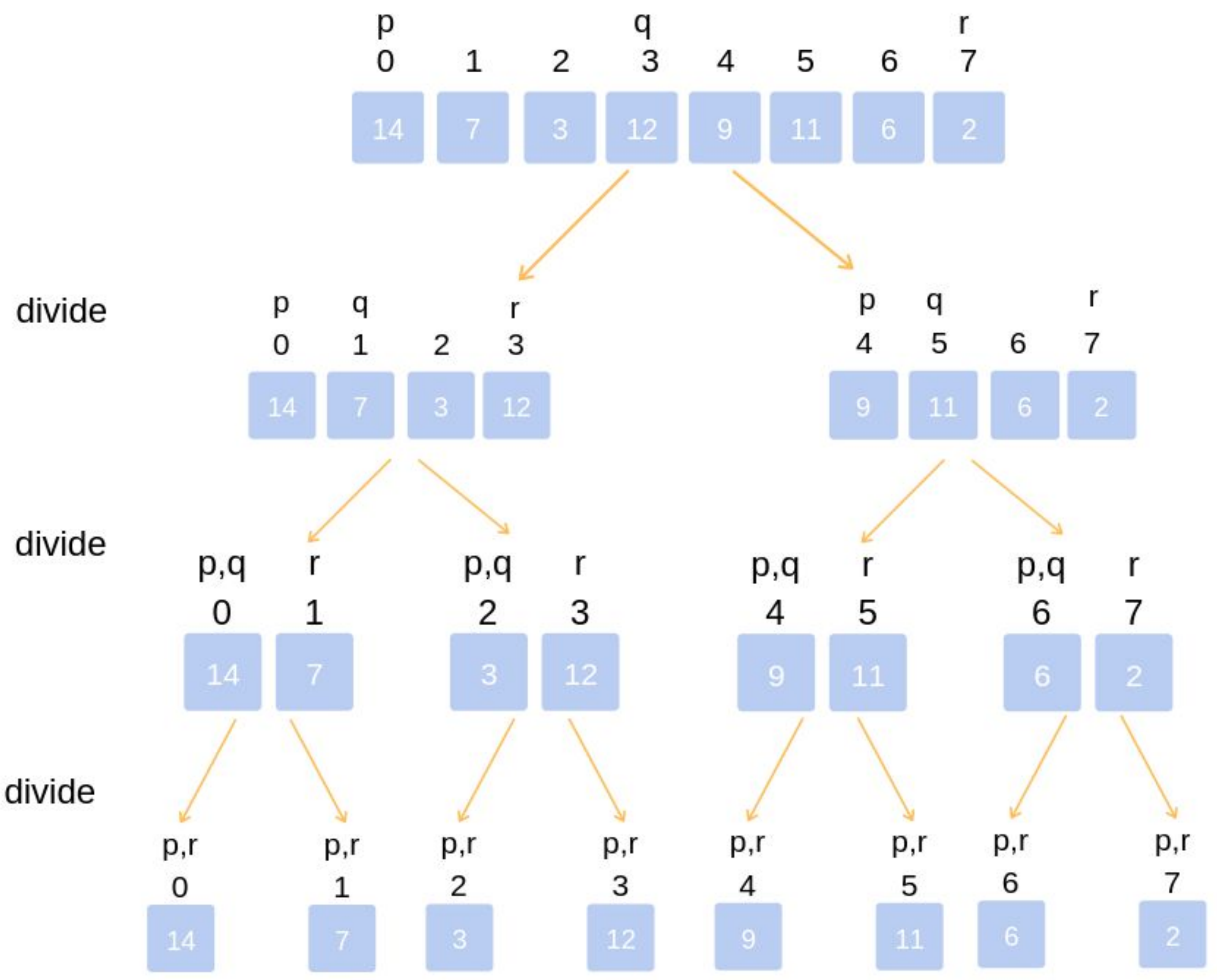


MergeSort (Example) - 22

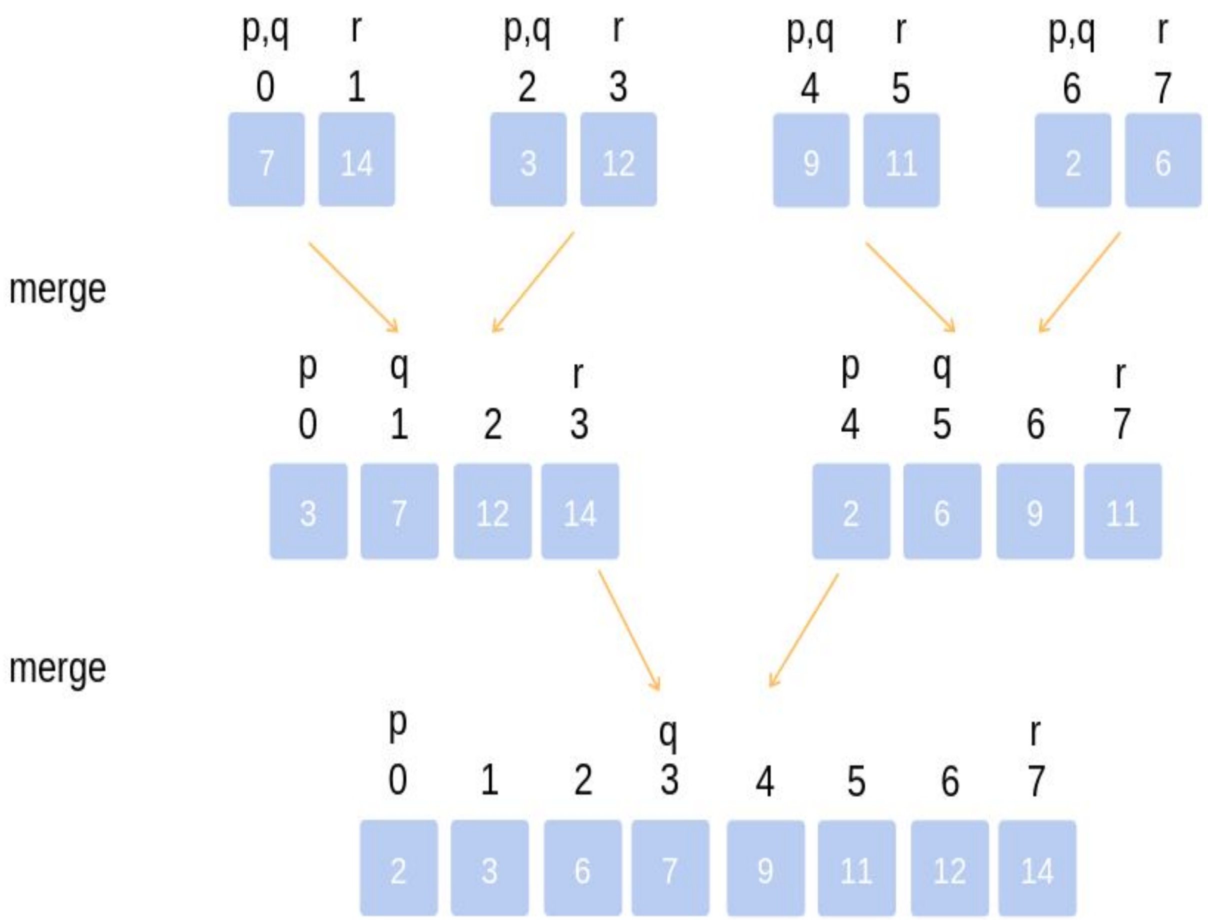


Merge Sort Algorithm

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 o
    {
        // 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)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The go
// 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 ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        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];
}
```



Analysis of Merge Sort

Worst Case:O(***nlogn***)

Averagevcade :O(***n logn***)

Quick Sort

- Quick sort, also known as partition sort, sorts by employing a **divide-and-conquer** strategy.
- Algorithm:
  - Pick an pivot element from the input.
  - Partition all other input elements such that elements less than the pivot come before the pivot and those greater than the pivot come after it (equal values can go either way).
  - Recursively sort the list of elements before the pivot and the list of elements after the pivot.
  - The recursion terminates when a list contains zero or one element.
- Time complexity: O($n$log$n$) or O(n^2)
- Example: Sort the list {25, 57, 48, 37, 12}

Quick sort I

To sort a[left...right]:

1. if left < right:

1.1. Partition a[left...right] such that:
all a[left...p-1] are less than a[p], and
all a[p+1...right] are >= a[p]

1.2. Quicksort a[left...p-1]   //recursive call

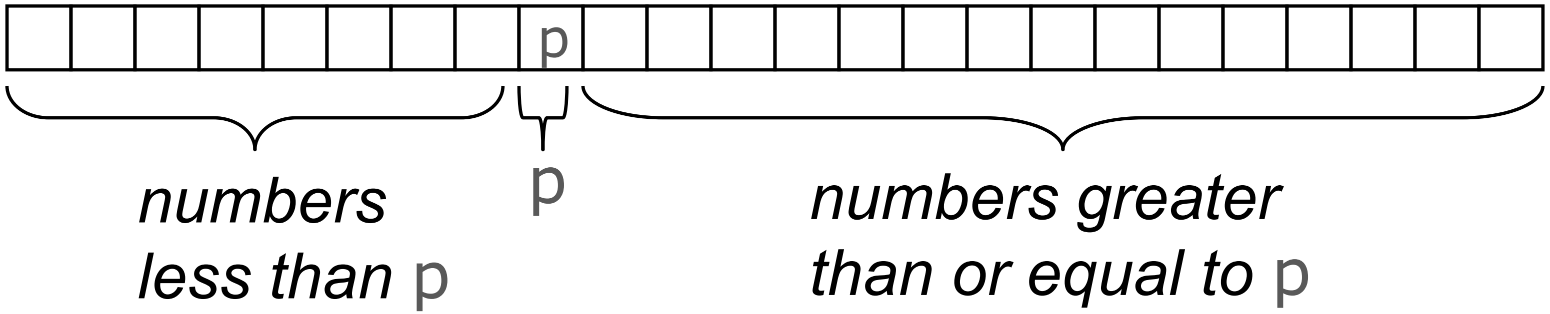
1.3. Quicksort a[p+1...right]

2. Terminate

Partitioning (Quick sort II)

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:



Partitioning II

- Choose an array value (say, the first) to use as the pivot
- Starting from the left end, find the first element that is greater than or equal to the pivot
- Searching backward from the right end, find the first element that is less than the pivot
- Interchange (swap) these two elements
- Repeat, searching from where we left off, until done

Partitioning

- To partition a[left...right]:

1. Set p = a[left], l = left + 1, r = right;
2. while l < r, do

2.1. while l < right & a[l] < p, set l = l + 1
2.2. while r >= left & a[r] >= p, set r = r - 1
2.3. if l < r, swap a[l] and a[r]

3. Set a[left] = a[r], a[r] = p
4. Terminate

Example of partitioning

choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
search: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6 (left > right)
swap with pivot: 1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

Analysis of quick sort

Cost of Quick Sort:

- three steps:
- Partition: has to compare (high-low) pairs
- first recursive call
- second recursive call

Analysis of quick sort (Best case)

We cut the array size in half each time

So the depth of the recursion is $\log_2 n$

At each level of the recursion, all the partitions at that level do work that is linear in n

$O(\log_2 n) * O(n) = O(n \log_2 n)$

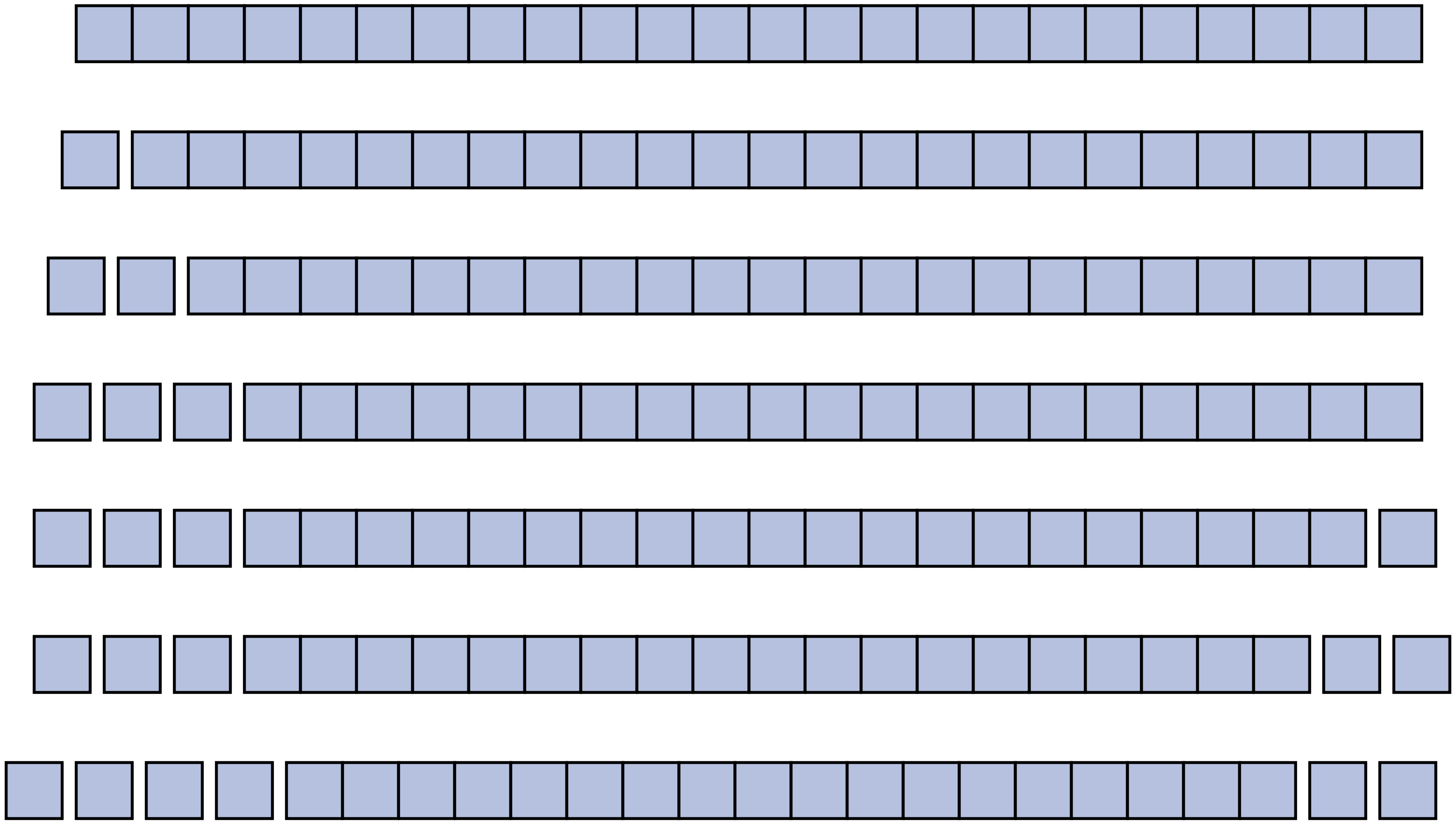
Hence in the average case, quick sort has time complexity $O(n \log_2 n)$

What about the worst case?

Worst case

- In the worst case, partitioning always divides the size `n` array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length `n-1` part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length `n-1` part requires (in the worst case) recurring to depth `n-1`

Worst case partitioning



Worst case for quick sort

In the worst case, recursion may be n levels deep (for an array of size n)

But the partitioning work done at each level is still n

$O(n) * O(n) = O(n^2)$

So worst case for Quick sort is $O(n^2)$

When does this happen?

- When the array is sorted to begin with!

Typical case for quick sort

- If the array is sorted to begin with, Quick sort is terrible: $O(n^2)$
- It is possible to construct other bad cases
- However, Quick sort is *usually* $O(n \log_2 n)$
- The constants are so good that Quick sort is generally the fastest algorithm known
- Most real-world sorting is done by Quick sort

Picking a better pivot

Before, we picked the *first* element of the sub-array to use as a pivot

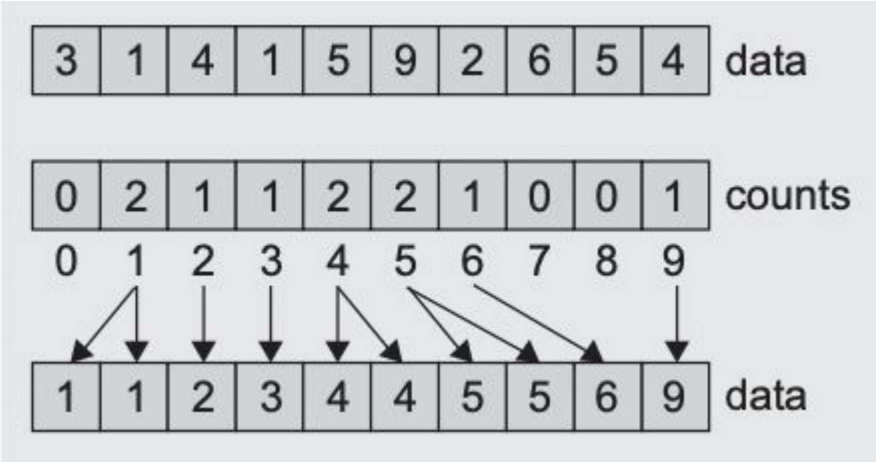
- If the array is already sorted, this results in $O(n^2)$ behavior
- It's no better if we pick the *last* element

We could do an *optimal* quick sort (guaranteed $O(n \log n)$) if we always picked a pivot value that exactly cuts the array in half

- Such a value is called a median: half of the values in the array are larger, half are smaller
- The easiest way to find the median is to *sort* the array and pick the value in the middle (!)

Bucket Sort

1. Bucket sort is possibly the simplest distribution sorting algorithm
2. In bucket sort, initially, a fixed number of buckets are selected
3. The bucket sort uses m buckets or counters.
4. The i^{th} counter/bucket keeps track of the number of occurrences of the i^{th} element of the list.
5. Here m can have integer numbers between [0..m-1];
6. Example for m= 10



Bucket Sort

```
void BucketSort(int A[], int n)
{
   int i, j;
   int bucket[max];
   //counters/buckets can store numbers maximum 20
   for(i = 0; i < max; i++)
      bucket[i] = 0;
   for(j = 0; j < n; j++)
   {
      ++bucket[A[j]];
      // counting number for each bucket
   }
   for(i = 0, j = 0; i < max; i++)
      for(;bucket[i] > 0; --bucket[i])
      { A[j] = i; j++; }
}
```

Bucket Sort - Complexity Analysis

1. **Best Case:**

a. If the array elements are **uniformly distributed**, **bucket size** will almost be the **same** for all the buckets.
b. To create n buckets and scatter each element from the array, time complexity = O(n)
c. If we consider the buckets less than size of element then:
d. If we use Insertion sort to sort each bucket, time complexity = O(k)
e. **Hence, best case time complexity for bucket sort = O(n+k), where n = number of elements, and k = number of buckets**

Bucket Sort - Complexity Analysis

2. <u>Worst Case:</u>

a. If the array elements are **not uniformly distributed,** i.e., elements are concentrated within specific ranges
b. This will result in one or more buckets having more elements than other buckets, making bucket sort like any other sorting technique, where every element is compared to the other.
c. Time complexity increases even further if the elements in the array are present in the reverse order
d. If insertion sort is used, the worst-case time complexity can go up to **O(n^2).**

Bucket Sort - Complexity Analysis

Bucket Sort time complexity

- Best Case Time Complexity: O(n+k)
- Average Case Time Complexity: O(n)
- Worst Case Time Complexity: O(n^2)

Bucket Sort - Space Complexity Analysis

- **Space Complexity : O(n+k)**
- where **n = number of elements in the array**, and **k = number of buckets** formed

  Space taken by each bucket is O(k), and inside each bucket, we have n elements **scattered**.
- Hence, the space complexity becomes O(n+k).

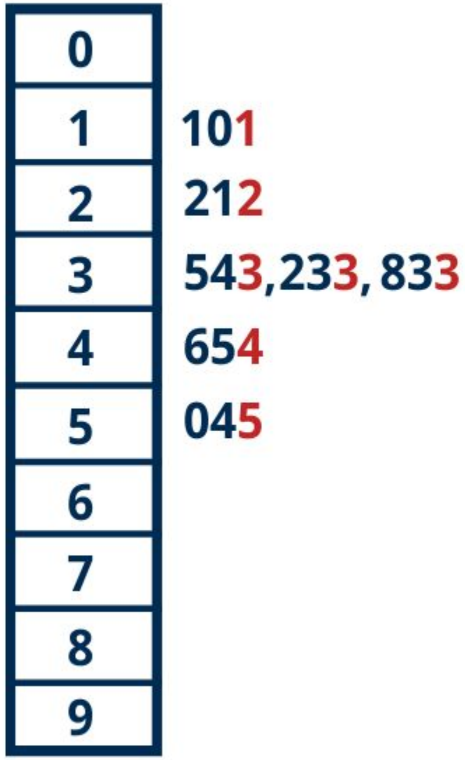
Radix Sort

- Radix sort is a generalization of Bucket sort.
- It works in following steps.
  - Distribute all elements into m buckets where m is an integer. For example if m is 10. We take 10 buckets numbered as 0,1,2,……..,9. for sorting strings, we may need 26 bucket, and so on.
  - Number of passes required to sort is equal to number of digits in the largest number in the list.
  - In the first pass, numbers are sorted on least significant digit. Numbers with the same least significant digit are stored in the same bucket.
  - In the 2nd pass, numbers are sorted on the second least significant digit.
  - At the end of every pass, numbers in buckets are merged to produce a common list.

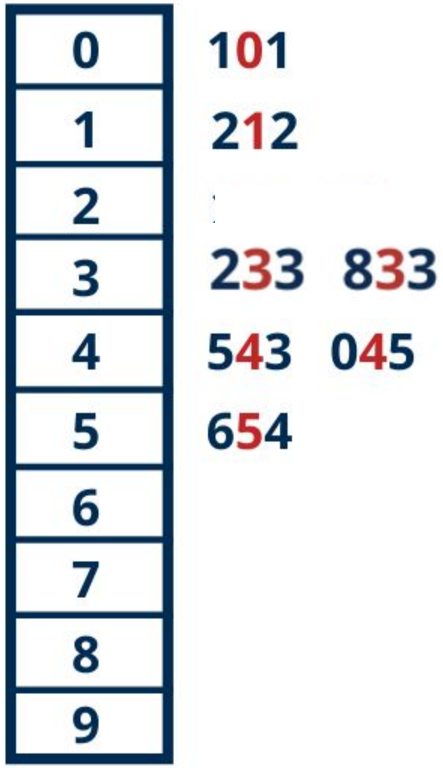
101	45	543	233	212	654	833

101	212	543	233	833	654	45

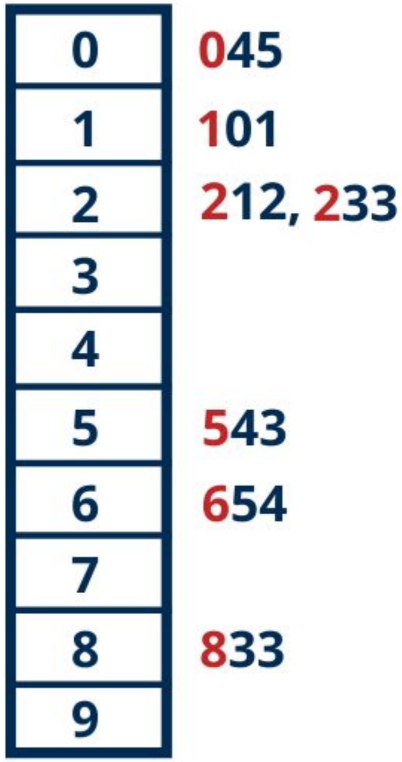
101	212	233	833	543	045	654



Unit place sorting in radix sort



10th place sorting in radix sort



100th place sorting in radix sort

Radix sort Example

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

Queue-0	Queue-1	Queue-2	Queue-3	Queue-4	Queue-5	Queue-6	Queue-7	Queue-8	Queue-9

Pass - 1

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8**2**, 90**1**, 10**0**, 1**2**, 15**0**, 7**7**, 5**5** & 2**3**

Queue-0	Queue-1	Queue-2	Queue-3	Queue-4	Queue-5	Queue-6	Queue-7	Queue-8	Queue-9
150 100	901	12 82	23		55		77		

Pass - 2

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1**0**0, 1**5**0, 9**0**1, **8**2, **1**2, **2**3, **5**5 & **7**7

Queue-0	Queue-1	Queue-2	Queue-3	Queue-4	Queue-5	Queue-6	Queue-7	Queue-8	Queue-9
901 100	12	23			55 150		77	82	

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Pass - 3

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, **9**01, 12, 23, **1**50, 55, 77 & 82

Queue-0	Queue-1	Queue-2	Queue-3	Queue-4	Queue-5	Queue-6	Queue-7	Queue-8	Queue-9
82 77 55 23 12	150 100								901

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

Pseudocode for Radix Sort

```
void sorting::radixtsort()
{
  int  large, pass, div, bktno, i,j,k, count[10], bkt[20][20];
  large=a[0];
  for(i=0;i<n;i++)
  {
       if(a[i]>large)
          {
   large=a[i];
          }
  }
    pass=0;
  while(large>0)
  {
       pass++;
       large=large/10;
  }
div=1;
for(i=1;i<=pass; i++)
{
  for(j=0;j<=9;j++)
   {
          count[j]=0;
   }
  for(j=0;j<n;j++)
  {
  bktno=(a[j]/div)%10;
  bkt[bktno][count[bktno]]=a[j];
  count[bktno]++;
   }
  j=0;
  for(bktno=0;bktno<=9;bktno++)
  {
      for(k=0;k<count[bktno];k++)
          {
                 a[j]=bkt[bktno][k];
                 j++;
          }
  }

  div=div*10;
 }
}
```

COMPARISON OF ALL SORTING METHODS

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable	Pros	Cons
Bubble sort	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. A simple and easy method 2. Efficient for small lists $n > 100$	Highly inefficient for large data

Selection sort	Finds the minimum value in the list and then swaps it with the value in the first position, repeats these steps for the remainder of the list (starting at the second position and advancing each time)	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. Recommended for small files 2. Good for partially sorted data	Inefficient for large lists

Insertion sort	Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already sorted list until no input elements remain. The choice of which element to remove from the input is arbitrary	$O(n)$	$O(n^2)$	No extra space needed	Yes	1. Relatively simple and easy to implement 2. Good for almost sorted data	Inefficient for large lists

Merge sort	Conceptually, a merge sort works as follows: If the list is of length 0 or 1, then it is already sorted. Otherwise, the algorithm divides the unsorted list into two sub-lists of about half the size Then, it sorts each sub-list recursively by reapplying the merge sort and then merges the two sub-lists back into one sorted list	$O(n\log_2 n)$	$O(n\log_2 n)$	Extra space proportional to n is needed	Yes	1. Good for external file sorting 2.Can be applied to files of any size	1. It requires twice the memory of the heap sort because of the second array used to store the sorted list. 2. It is recursive, which can make it a bad choice for applications that run on machines with limited memory

Practice Problem Statement

1.Given a sorted array **Arr[]**(0-index based) consisting of **N** distinct integers and an integer **k**, the task is to find the index of k, if it's present in the array **Arr[]**. Otherwise, find the index where **k** must be inserted to keep the array sorted.

2.There are n tree in a forest. Heights of the trees is stored in array tree[ ]. If ith tree is cut at height h, the wood obtained is tree[i]-h, given that tree[i]>h. If total wood needed is k (not less, neither more) find the height at which every tree should be cut (all trees have to be cut at the same height).

3.Given an integer **K** and an array **height[]** where **height[i]** denotes the height of the **ith** tree in a forest.The task is to make a cut of height **X** from the ground such that exactly **K** unit wood is collected.If it is not possible then print **-1** else print **X**.

Practice Problem Statement

4. Given a sorted array with possibly duplicate elements, the task is to find indexes of first and last occurrences of an element x in the given array.

5.Given an array of integers. Find a peak element in it. An array element is a peak if it is NOT smaller than its neighbours. For corner elements, we need to consider only one neighbour.

6.Given a sorted and rotated array **A** of N distinct elements which is rotated at some point, and given an element **K.** The task is to find the index of the given element K in the array A.

Practice Problem Statement

7.Given two arrays **A** and **B** contains integers of size **N** and **M**, the task is to find numbers which are present in the first array, but not present in the second array.

8.Given an integer **x,** find the square root of x. If **x** is not a perfect square, then return floor(√x).

Practice Problem Statement

Given an array of distinct integers. Sort the array into a wave-like array and return it. In other words, arrange the elements into a sequence such that a1 >= a2 <= a3 >= a4 <= a5..... (considering the increasing lexicographical order).

Input: n = 5
arr[] = {1,2,3, 4,5 }
Output: 2 1 4 3 5
Explanation: Array elements after sorting it in wave form are 2 1 4 3 5.

Practice Problem Statement

Given two integer arrays **A1[ ]** and **A2[ ]** of size **N** and **M** respectively. Sort the first array **A1[ ]** such that all the relative positions of the elements in the first array are the same as the elements in the second array **A2[ ]**.

See example for better understanding.

Note: If elements are repeated in the second array, consider their first occurrence only.

Example 1:

Input:

N = 11

M = 4

A1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8}

A2[] = {2, 1, 8, 3}

Output:

2 2 1 1 8 8 3 5 6 7 9

Explanation: Array elements of A1[] are sorted according to A2[]. So 2 comes first then 1 comes, then comes 8, then finally 3 comes, now we append remaining elements in sorted order.

Given an array Arr[] of N distinct integers and a range from L to R, the task is to count the number of triplets having a sum in the range [L, R].

Input:
N = 4
Arr = {8 , 3, 5, 2}
L = 7, R = 11
Output: 1
Explanation: There is only one triplet {2, 3, 5} having sum 10 in range [7, 11].