Asymptotic Notation

O big-oh upper bound

Ω big-omega lower bound

gt is used $\longrightarrow$ θ theta Average bound

lower bound — avg. bound — upper bound

$$\boxed{1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 \ldots 2^n < 3^n \ldots < n^n}$$

⇒ Big-oh

The function $f(n) = O(g(n))$ iff $\exists$ +ve Constants c and n_0

such that $f(n) \leq c * g(n) \; \forall \; n \geq n_0$

e.g. $f(n) = 2n+3$

$$2n+3 \leq \underbrace{10}_{c}\underbrace{n}_{g(n)} \quad n \geq 1$$

$\hookrightarrow$ gt can be any no. but greater than $2n+3$.

$\boxed{\therefore f(n) = O(n)}$ $\uparrow$ This is useful.

or

$2n+3 \leq ?$

$2n+3 \leq 2n+3n$

$2n+3 \leq 5n \quad n \geq 1$

$f(n) = O(n)$

or

eg- $f(n) = 2n+3$

$2n+3 \leq 2n^2 + 3n^2$

$f(n) \leftarrow 2n \leq 5n^2 \quad n \geq 1 \quad c\,g(n)$

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

Scanned with CamScanner

Omega

The function $f(n) = \Omega(g(n))$ iff $\exists$ +ve Constant C and n_0.
Such that $f(n) \geq c * g(n) \ \forall \ n \geq n_0$.

e.g. $f(n) = 2n+3$

$2n+3 \geq 1 \times \log n \ \forall \ n \geq 1$

$\underset{f(n)}{\downarrow}$ $\quad \overset{\uparrow}{c} \quad \overset{\uparrow}{g(n)}$.

$\boxed{\therefore f(n) = \Omega(n)}$
$f(n) = \Omega \log(n$
→ This is useful.

~~⊖~~ Note : If we write other variable it may be true but not useful.

Theta Notation.

The function $f(n) = \theta(g(n))$ iff $\exists$ +ve Constants C_1, C_2 and n_0.
such that $c_1 * g(n) \leq f(n) \leq C_2 * g(n)$

e.g. $f(n) = 2n+3$

$1 \times n \leq 2n+3 \leq 5 \times n$
$c_1 \, g(n) \quad f(n) \quad C_2 \, g(n)$.

$\therefore \underbrace{f(n) = \theta(n)}$
↓
This only things that is possible.

Note : It is mostly recomend.

Que.1 $f(n) = 2n^2 + 3n + 4$.

(i) $2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$.

$2n^2 + 3n + 4 \leq 9n^2$. $\boxed{n \geq 1}$

$$\underset{\substack{\uparrow \\ c}}{9}\,\underset{\substack{\uparrow \\ g(n)}}{n^2}$$

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

(ii) ~~f(n)~~ $2n^2 + 3n + 4 \geq 1 \times n^2$.

$\Omega(n^2)$

(iii) $1 \times n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$ $\Theta(n^2)$

Que.2. $f(n) = n^2 \log n + n$.

$1 \times n^2 \log n \leq n^2 \log n + n \leq 10 n^2 \log n$.

$O(n^2 \log n)$ $\Omega(n^2 \log n)$ $\Theta(n^2 \log n)$

Que.3 $f(n) = n! = n \times (n-1) \times (n-2) \text{---} 3 \times 2 \times 1$

$1 \times 1 \times \times \text{---} \times 1 \leq 1 \times 2 \times 3 \text{---} \times n \leq n \times n \times n \text{--} \times n$

$1 \leq n! \leq n^n$

$\Omega(1)$ $O(n^n)$

↳ Here we can not find any

↳ for factorial ~~we~~ function we go for lower bound and upper bound.

Scanned with CamScanner

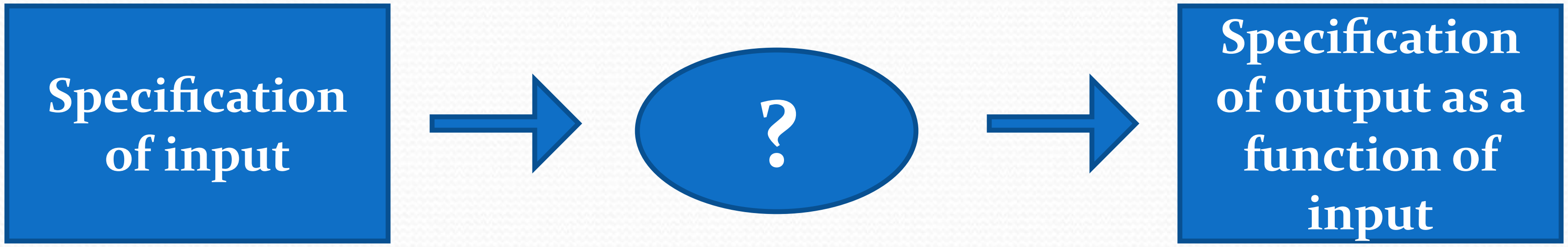
DATA STRUCTURES

- Introduction:
  - Basic Concepts and Notations
  - Complexity analysis: time space tradeoff
  - Algorithmic notations, Big O notation
  - Introduction to omega, theta and little o notation

Basic Concepts and Notations

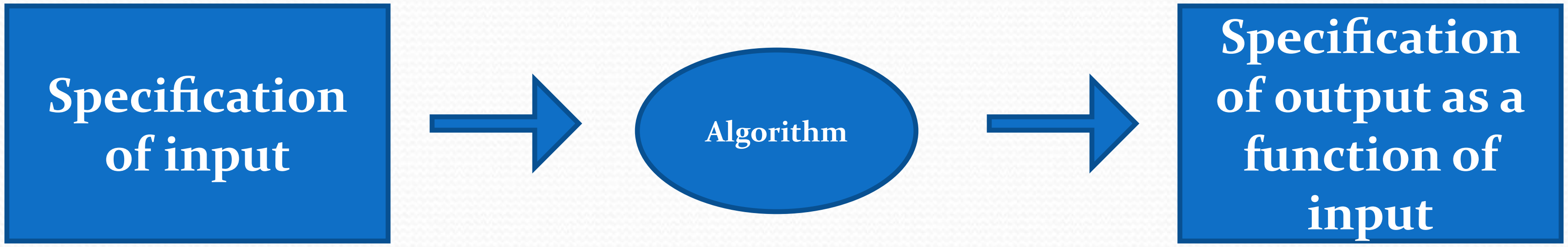
- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions
- Program: an implementation of an algorithm in some programming language
- Data Structure: **Organization** of data needed to solve the problem

Algorithmic Problem



- Infinite number of input instances satisfying the specification. For example: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length:
  - 1, 20, 908, 909, 100000, 1000000000.
  - 3.

Algorithmic Solution



- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

What is a Good Algorithm?

- Efficient:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - The number of bits in an input number
  - Number of data elements(numbers, points)

Complexity Analysis and Time Space Trade-off

Complexity

- A measure of the performance of an algorithm
- An algorithm’s performance depends on
  - *internal* factors
  - *external* factors

External Factors

- Speed of the computer on which it is run
- Quality of the compiler
- Size of the input to the algorithm

Internal Factor

- **The algorithm's efficiency, in terms of:**
- **Time required to run**
- **Space (memory storage)required to run**

- **Note:**
- **Complexity measures the *internal* factors (usually more interested in time than space)**

Two ways of finding complexity

- Experimental study
- Theoretical Analysis

Experimental study

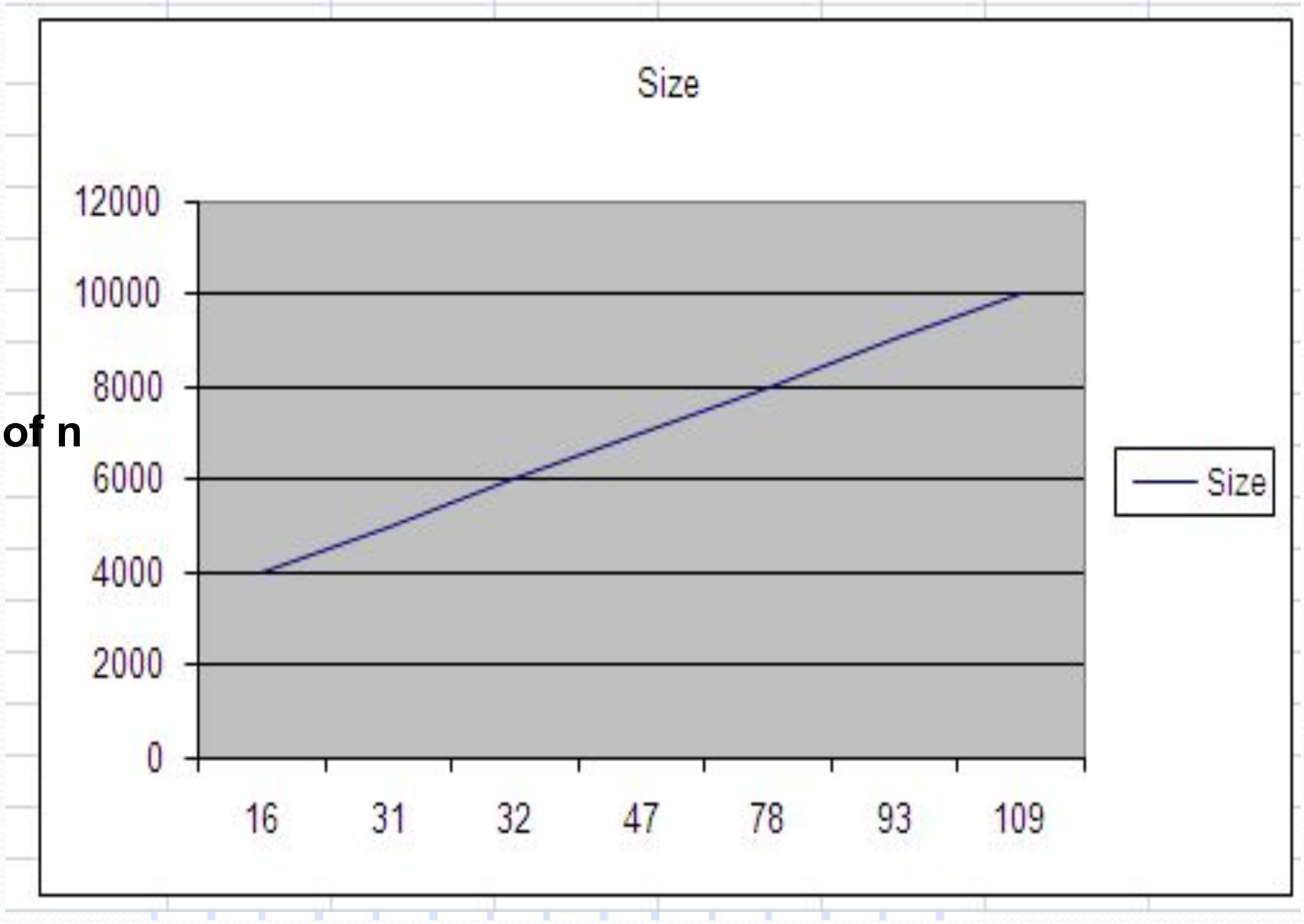
- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time

  Use a method like System.currentTimeMillis()
- Plot the results

Example

- a.
```
Sum=0;
for(i=0;i<N;i++)
  for(j=0;j<i;j++)
      Sum++;
```

Example graph



Size of n

- Time in millisec

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Complexity analysis

- Why we should analyze algorithms?
  - Predict the resources that the algorithm requires
    - Computational time (CPU consumption)
    - Memory space (RAM consumption)
    - Communication bandwidth consumption
  - The running time of an algorithm is:
    - The total number of primitive operations executed (machine independent steps)
    - Also known as algorithm complexity

Need for analysis : Internal Factors

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

Space Complexity

- The space needed by an algorithm is the sum of a fixed part and a variable part
- The fixed part includes space for
  - Instructions
  - Simple variables
  - Fixed size component variables
  - Space for constants
  - Etc..

Cont…

- The variable part includes space for
  - Component variables whose size is dependant on the particular problem instance being solved
  - Recursion stack space
  - Etc..

Time Complexity

- The time complexity of a problem is
  - the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm.
- The exact number of steps will depend on exactly what machine or language is being used.
- To avoid that problem, the Asymptotic notation is generally used.

Time Complexity

- Worst-case
  - An upper bound on the running time for any input of given size
- Average-case
  - Assume all inputs of a given size are equally likely
- Best-case
  - The lower bound on the running time

Time Complexity – Example

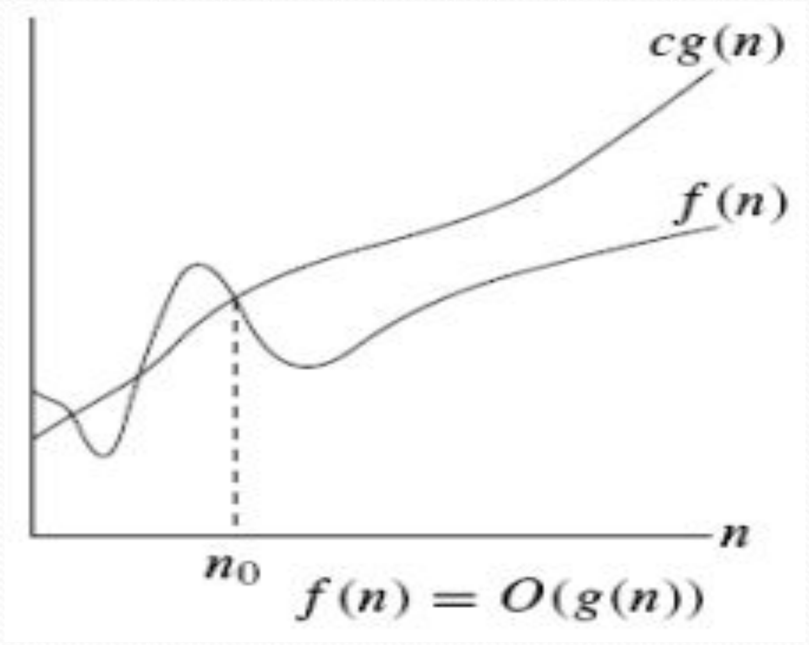
- Sequential search in a list of size n
  - Worst-case:
    - n comparisons
  - Best-case:
    - 1 comparison
  - Average-case:
    - n/2 comparisons

Asymptotic notations

- Algorithm complexity is rough estimation of the number of steps performed by given computation depending on the size of the input data
  - Measured through **asymptotic notation**
    - `O(g)` where `g` is a function of the input data size
  - Examples:
    - Linear complexity `O(n)` – all elements are processed once (or constant number of times)
    - Quadratic complexity $O(n^2)$ – each of the elements is processed `n` times

O-notation

Asymptotic upper bound



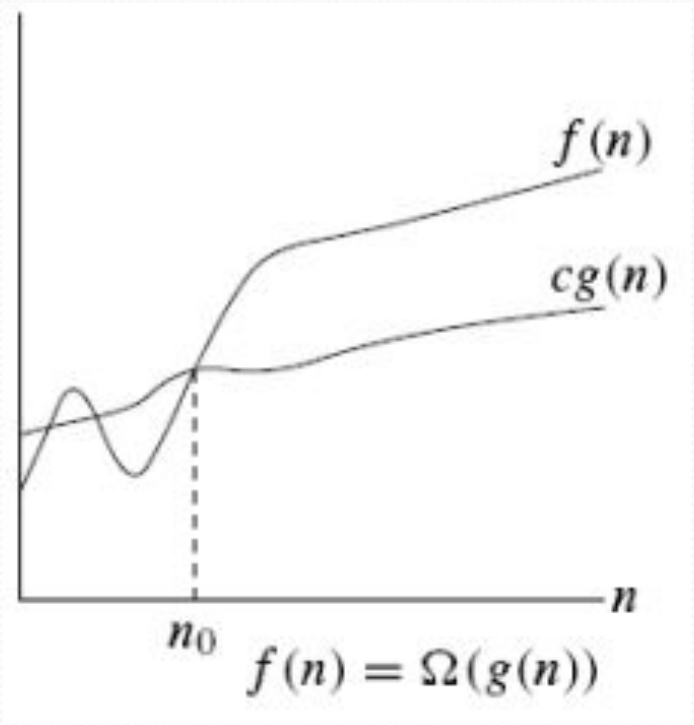
$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

Example

- The running time is $O(n^2)$ means there is a function f(n) that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time of that input is bounded from above by the value f(n).
  - $3 * n^2 + n/2 + 12 \in O(n^2)$
  - $4*n*\log_2(3*n+1) + 2*n-1 \in O(n * \log n)$

Ω notation

Asymptotic lower bound



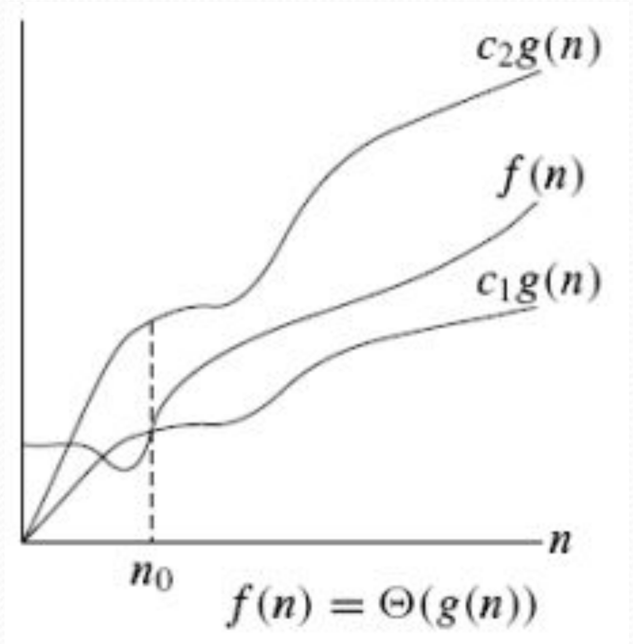
$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}\ .$$

Example

- When we say that the running time (no modifier) of an algorithm is Ω (g(n)).
- we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times g(n), for sufficiently large n.
- $n^3 + 20n \in \Omega(n^2)$

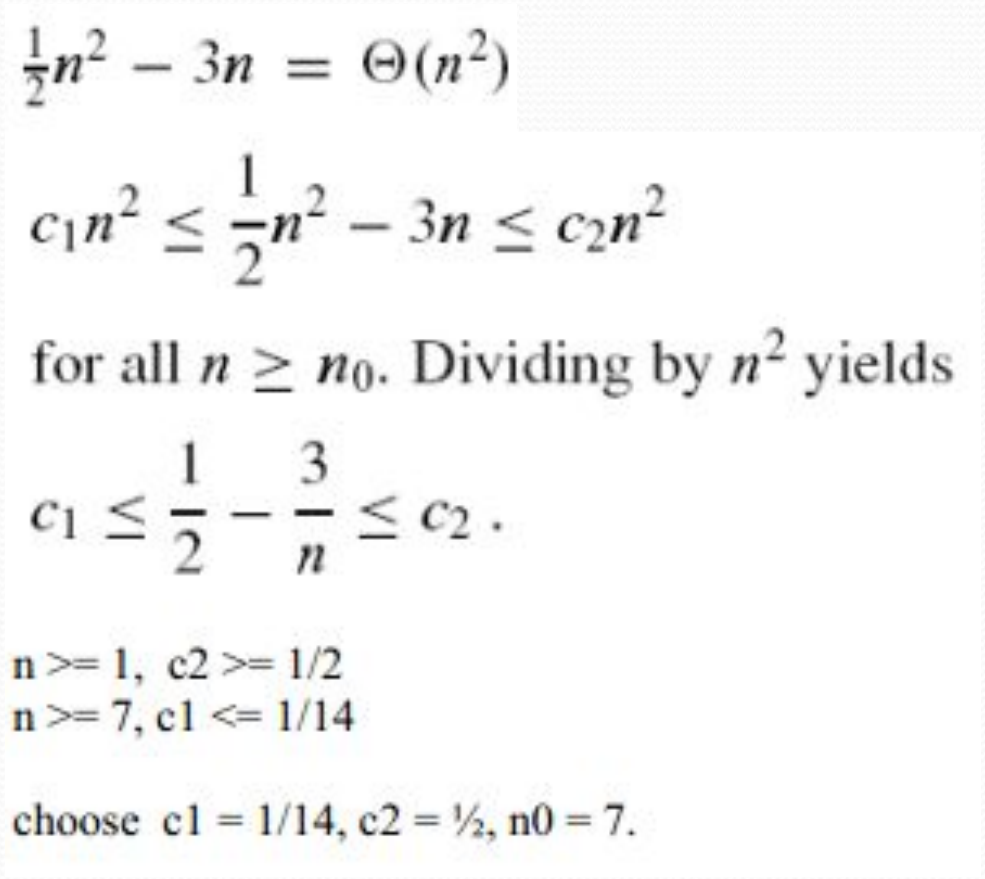
Θ notation

g(n) is an asymptotically tight bound of f(n)



$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.[1]

Example



$\frac{1}{2}n^2 - 3n = \Theta(n^2)$

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

for all $n \ge n_0$. Dividing by n^2 yields

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2 \, .$$

n >= 1, c2 >= 1/2
n >= 7, c1 <= 1/14

choose c1 = 1/14, c2 = ½, n0 = 7.

Big O notation

- $f(n)=O(g(n))$ iff there exist a positive constant c and non-negative integer n_o such that

$$f(n) \leq cg(n) \quad \text{for all } n \geq no.$$

- $g(n)$ is said to be an upper bound of $f(n)$.

Basic rules

1. Nested loops are multiplied together.
2. Sequential loops are added.
3. Only the largest term is kept, all others are dropped.
4. Constants are dropped.
5. Conditional checks are constant (i.e. 1).

Example of complexity

Linear loop

1)

```
for(int i = 0; i < 10; i++)
    {
        cout << i << endl;
    }
```

//time taken = ?

2)

```
  for(int i = 0; i < n; i++)
  {
    cout << i << endl;
  }
```

//time taken = ?

- Ans: O(n)

Quadratic Loops

```
1) for(int i = 0; i < 100; i++)
    {
        for(int j = 0; j < 100; j++)
         {
        //do swap stuff, constant time
        }
    }      //Time Taken =?
```

```
2) for(int i = 0; i < n; i++)
   {
      for(int j = 0; j < n; j++)
       {
      //do swap stuff, constant time
      }
   }          //Time Taken =?
```

- Ans O(n^2)

Complex condition

1)
```
for(int i = 0; i < 2*100; i++)
    {
   cout << i << endl;
   }
```

//Time Taken =?

```
2) for(int i = 0; i < 2*n; i++)
     {
    cout << i << endl;
    }
```

//Time Taken =?

- At first you might say that the upper bound is O(2n); however, we drop constants so it becomes O(n)

More loops in one program

1)
```
for(int i = 0; i <10 ; i++)
{
    cout << i << endl;
   }

for(int i = 0; i < 100; i++)
   {
   for(int j = 0; j < 100; j++)
       {
       //do constant time stuff
       }
   } //Time Taken =?
```

2)

```
for(int i = 0; i < n; i++)
      {
           cout << i << endl;
          }

      for(int i = 0; i < n; i++)
           {
           for(int j = 0; j < n; j++)
                {
                //do constant time stuff
                }
           } //Time Taken =?
```

- Ans : In this case we add each loop's Big O, in this case n+n^2. O(n^2+n) is not an acceptable answer since we must drop the lowest term. The upper bound is O(n^2). Why? Because it has the largest growth rate

Quadratic loop

```
1) for(int i = 0; i < 100; i++)
   {
       for(int j = 0; j < 2; j++)
       {
       //do stuff
       }
   } //Time Taken =?
```

```
2) for(int i = 0; i < n; i++)
   {
       for(int j = 0; j < 2; j++)
       {
       //do stuff
       }
   }
```

//Time Taken =?

- Ans: Outer loop is 'n', inner loop is 2, this we have 2n, dropped constant gives up O(n)

Complex iteration

```
1) for(int i = 1; i < n; i =i* 2)
   {
      cout << i << endl;
   }
```

//Time Taken =?

```
2) for(int i = 1; i < 100; i =i* 2)
   {
      cout << i << endl;
   }
```

//Time Taken =?

- There are n iterations, however, instead of simply incrementing, 'i' is increased by 2*itself each run. Thus the loop is log(n).

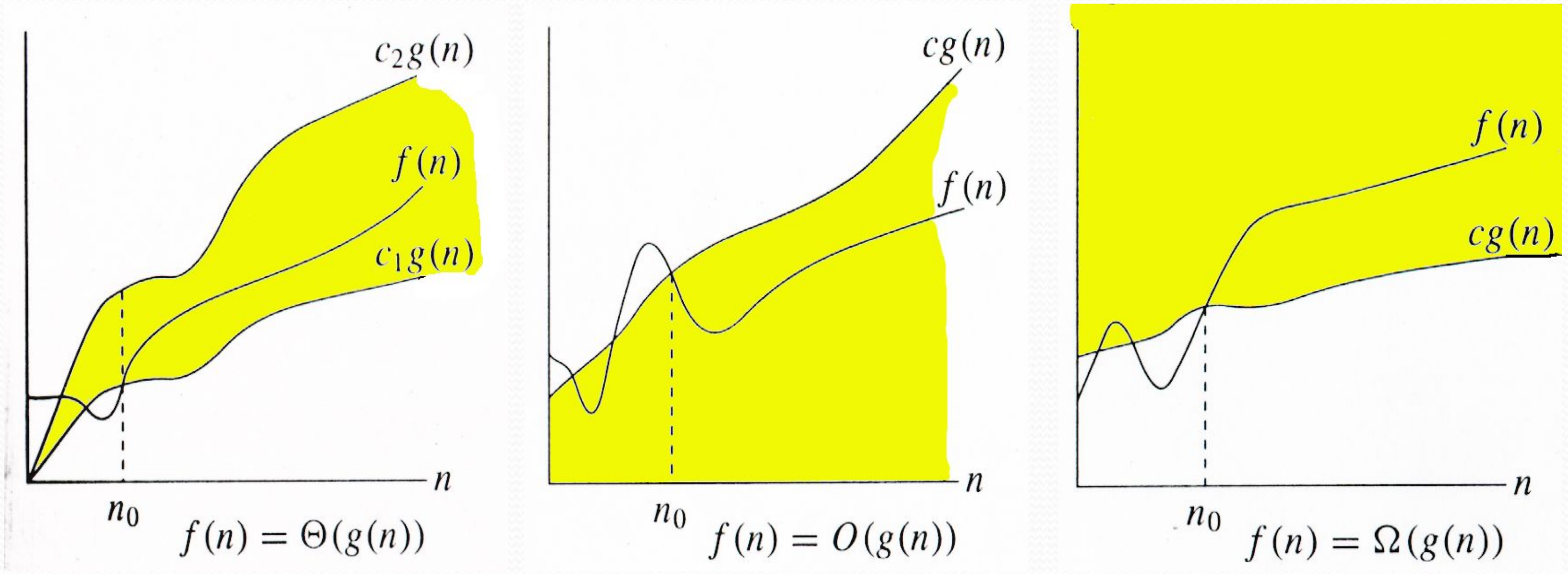
Quadratic loop

```
1) for(int i = 0; i < n; i++)
    {
        for(int j = 1; j < n; j *= 2)
            {
            //do constant time stuff
            }
    }
//Time Taken =?
```

- Ans: n*log(n)

```
While (n>=1)
{
    n=n/2;
}

2) While (n>=1)
{
    n=n/2;
}
```

Relations Between Θ, *O*, Ω



time space tradeoff

- A time space tradeoff is a situation where the memory use can be reduced at the cost of slower program execution (and, conversely, the computation time can be reduced at the cost of increased memory use).
- As the relative costs of CPU cycles, RAM space, and hard drive space change—hard drive space has for some time been getting cheaper at a much faster rate than other components of computers[citation needed]—the appropriate choices for time space tradeoff have changed radically.
- Often, by exploiting a time space tradeoff, a program can be made to run much faster.

Time Space Trade-off

- In computer science, a **space-time** or **time-memory trade off** is a situation where the memory use can be reduced at the cost of slower program execution (or, vice versa, the computation time can be reduced at the cost of increased memory use). As the relative costs of CPU cycles, RAM space, and hard drive space change — hard drive space has for some time been getting cheaper at a much faster rate than other components of computers-the appropriate choices for space-time tradeoffs have changed radically. Often, by exploiting a space-time tradeoff, a program can be made to run much faster.

Types of Time Space Trade-off

- **Lookup tables v. recalculation**

  The most common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

- **Compressed v. uncompressed data**

  A space-time trade off can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm). Depending on the particular instance of the problem, either way is practical.

Thank You

Searching & Sorting

Linear Search

Linear Search

- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every items is checked and if a match founds then that particular item is returned otherwise search continues till the end of the data collection.

Linear Search

- The sequential search (also called the linear search) is the simplest search algorithm.
- It is also the least efficient.
- It simply examines each element sequentially, starting with the first element, until it finds the key element or it reaches the end of the array.

  Example: If you were looking for someone on a moving passenger train, you would use a sequential search.

Algorithm

A linear array DATA with N elements and a specific ITEM of information are given. This algorithm finds the location LOC of ITEM in the array DATA or sets LOC = 0.

```
1. [Initialize] Set K := 1 and LOC := 0.
2. Repeat Steps 3 and 4 while LOC = 0 and K <= N.
3.      If ITEM = DATA[K], then: Set LOC := K.
4.      Set K := K + 1. [Increments counter.]
   [End of Step 2 loop.]
5. [Successful?]
     If Loc = 0, then:
               Write: ITEM is not in the array DATA.
          Else:
               Write: LOC is the location of the ITEM.
     [End of If structure.]
6. Exit.
```

Binary search

The Binary Search Algorithm

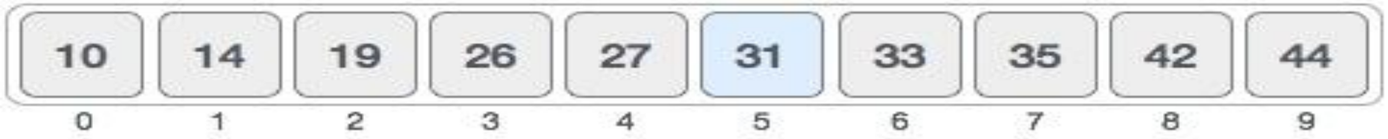
- The binary search is the standard algorithm for searching through a sorted sequence.
- It is much more efficient than the sequential search, but it does require that the elements be in order.
- It repeatedly divides the sequence in two, each time restricting the search to the half that would contain the element.
- You might use the binary search to look up a word in a dictionary.

The Binary Search Algorithm

- This search algorithm works on the principle of divide and conquer.
- For this algorithm to work properly the data collection should be in sorted form.
- Binary search search a particular item by comparing the middle most item of the collection.
- If match occurs then index of item is returned.
- If middle item is greater than item then item is searched in sub-array to the right of the middle item other wise item is search in sub-array to the left of the middle item.
- This process continues on sub-array as well until the size of subarray reduces to zero.

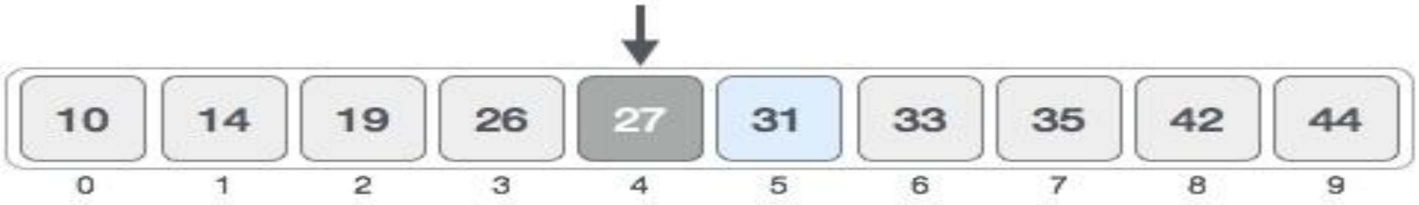
How binary search works?

- For a binary search to work, it is mandatory for the target array to be sorted.
- We shall learn the process of binary search with an pictorial example.
- The below given is our sorted array and assume that we need to search location of value 31 using binary search.



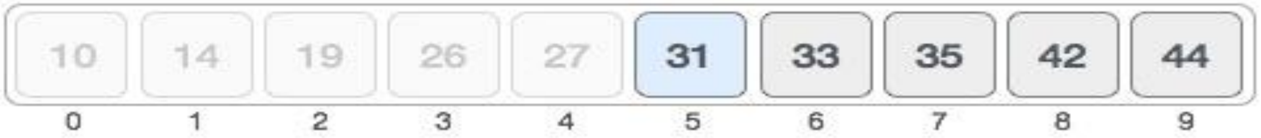
- First, we shall determine the half of the array by using this formula −

  mid = low + (high - low) / 2
- Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So 4 is the mid of array.



How binary search works?

- Now we compare the value stored at location 4, with the value being searched i.e. 31.
- We find that value at location 4 is 27, which is not a match. Because value is greater than 27 and we have a sorted array so we also know that target value must be in upper portion of the array.

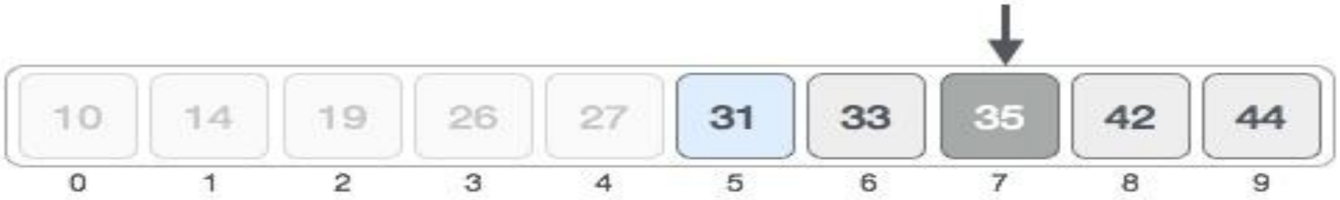


- We change our low to mid + 1 and find the new mid value again.

  low = mid + 1

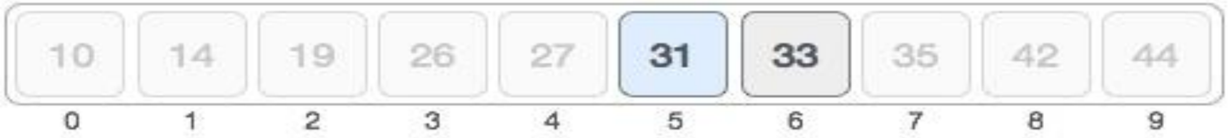
  mid = low + (high - low) / 2

- Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

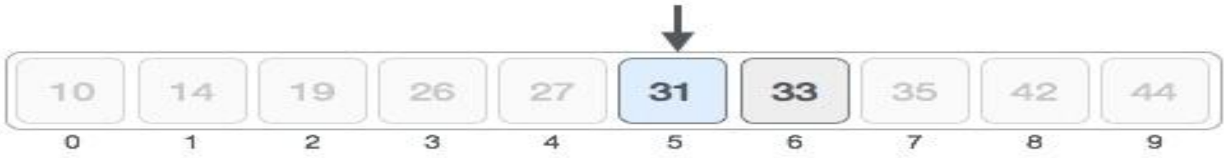


How binary search works?

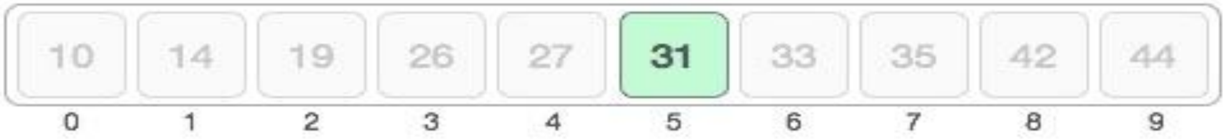
- The value stored at location 7 is not a match, rather it is less that what we are looking for. So the value must be in lower part from this location.



- So we calculate the mid again. This time it is 5.



- We compare the value stored ad location 5 with our target value. We find that it is a match.



- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

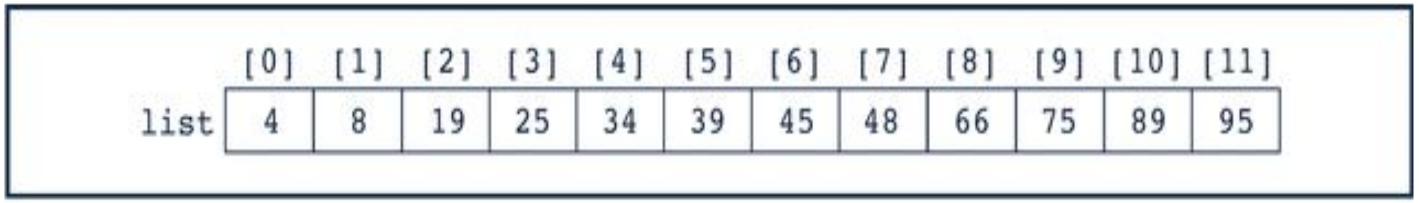
Algorithm

(Binary search) BINARY(DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, resp., the beginning, end, and middle locations of a segment of elements of DATA. The algo finds the location LOC of ITEM in DATA or sets LOC = NULL.

```
1.  [Initialize segment variables.]
    Set BEG := LB,   END := UB  and   MID := Int((BEG + END) / 2).
2.  Repeat steps 3 and 4 while BEG <= END  and  DATA[MID] != ITEM.
3.      If ITEM < DATA[MID], then:
            Set END := MID – 1.
        Else: Set BEG := MID + 1.
        [End of If.]
4.      Set MID := INT((BEG + END)/2).
    [End of Step 2 loop.]
5.  If DATA[MID] = ITEM, then:
        Set LOC := MID.
    Else: Set LOC := NULL.
    [End of If.]
6.  Exit.
```

Binary Search: Example



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

[redacted] Values of first, last, and middle and the Number of Comparisons for Search Item 89

Iteration	first	last	mid	list[mid]	[redacted]
1	0	11	5	39	[redacted]
2	6	11	8	66	[redacted]
3	9	11	10	89	[redacted]

Binary Search

[0]	ant
[1]	cat
[2]	chicken
[3]	cow
[4]	deer
[5]	dog
[6]	fish
[7]	goat
[8]	horse
[9]	camel
[10]	snake

Searching for cat

BinarySearch(0, 10)	middle: 5	cat < dog
BinarySearch(0, 4)	middle: 2	cat < chicken
BinarySearch(0, 1)	middle: 0	cat > ant
BinarySearch(1, 1)	middle: 1	cat = cat **Return: true**

Searching for zebra

BinarySearch(0, 10)	middle: 5	zebra > dog
BinarySearch(6, 10)	middle: 8	zebra > horse
BinarySearch(9, 10)	middle: 9	zebra > camel
BinarySearch(10, 10)	middle: 10	zebra > snake
BinarySearch(11, 10)		last > first **Return: false**

Searching for fish

BinarySearch(0, 10)	middle: 5	fish > dog
BinarySearch(6, 10)	middle: 8	fish < horse
BinarySearch(6, 7)	middle: 6	fish = fish **Return: true**

Sorting Techniques

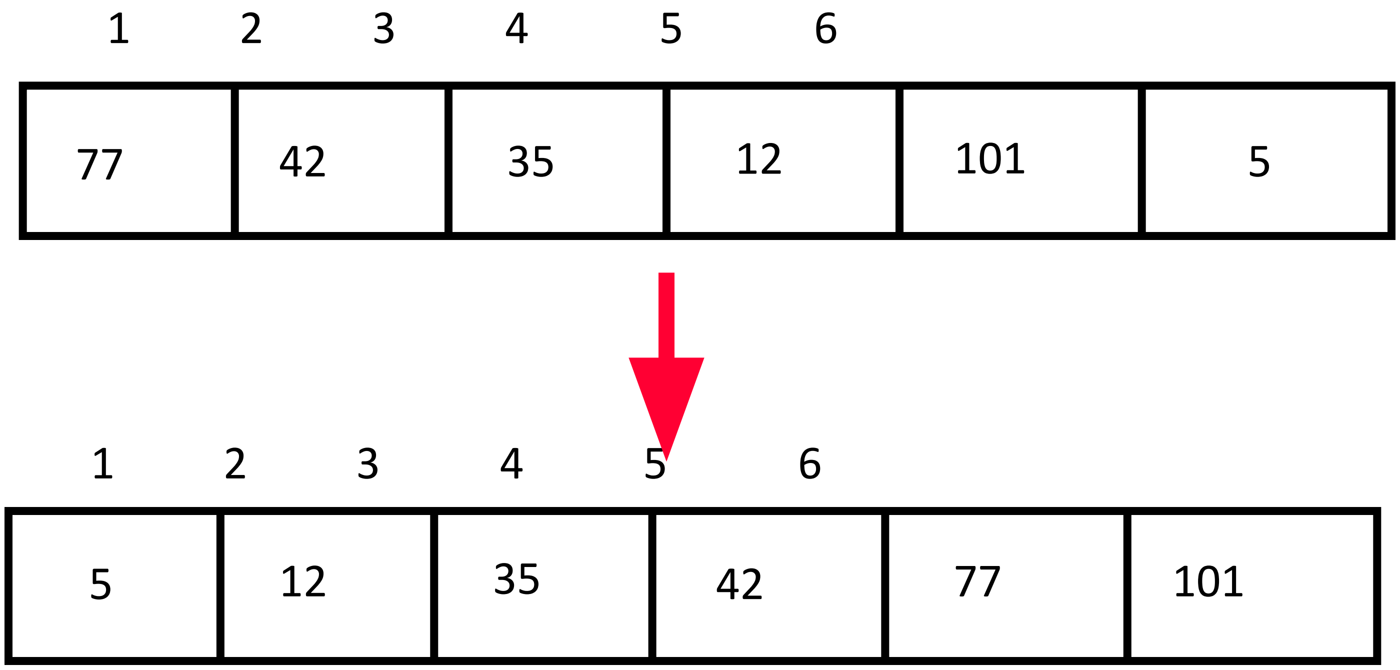
- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are numerical or lexicographical order.
- Importance of sorting lies in the fact that data searching can be optimized to a very high level if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Some of the examples of sorting in real life scenarios are following:

Telephone Directory − Telephone directory keeps telephone no. of people sorted on their names. So that names can be searched.

Dictionary − Dictionary keeps words in alphabetical order so that searching of any work becomes easy.

Example

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



Bubble Sort

Bubble Sort

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order.

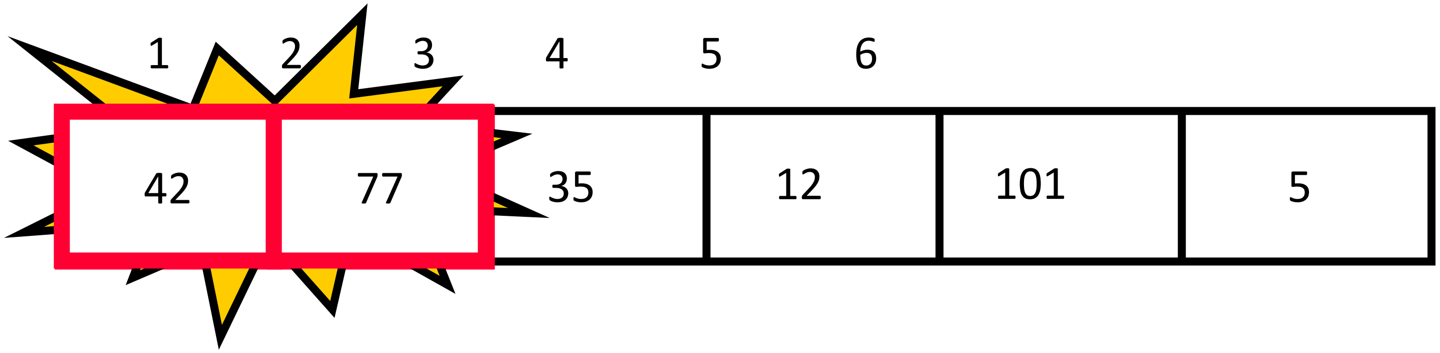
"Bubbling Up" the Largest Element

- **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

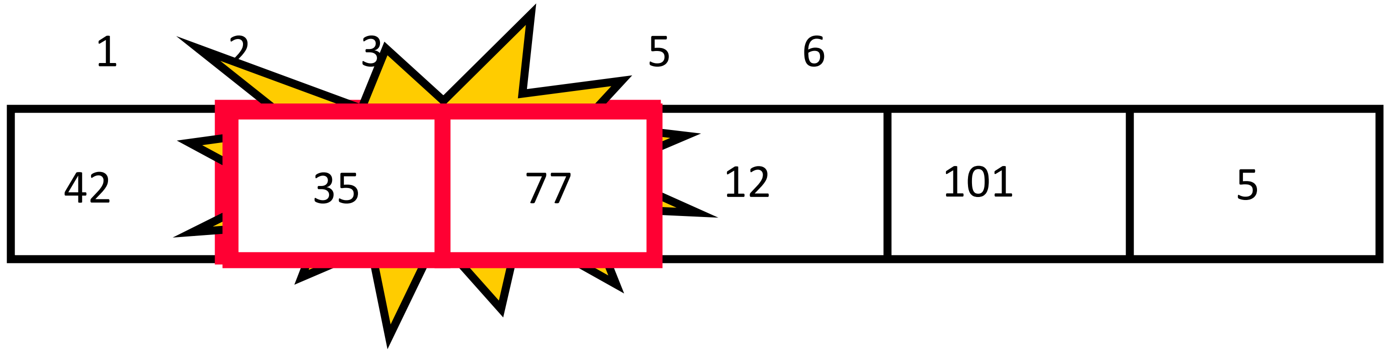
"Bubbling Up" the Largest Element

- 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



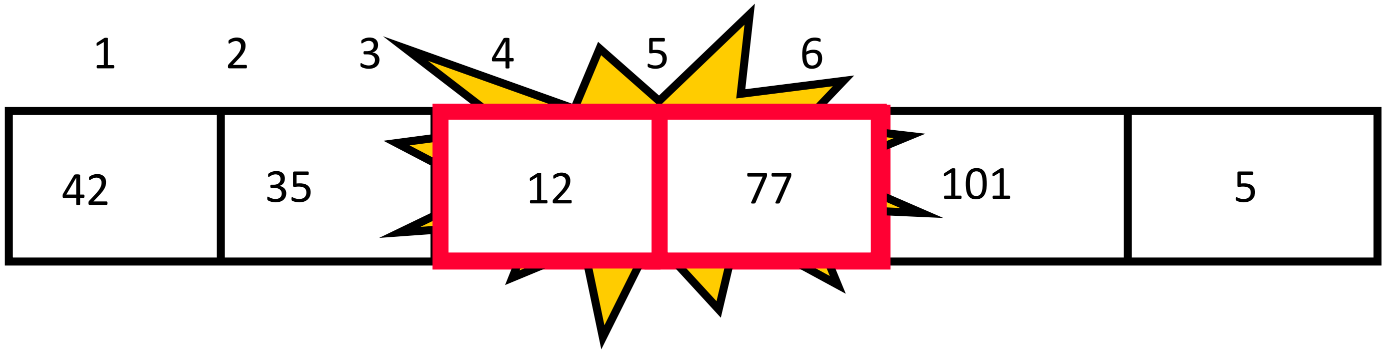
"Bubbling Up" the Largest Element

- **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**



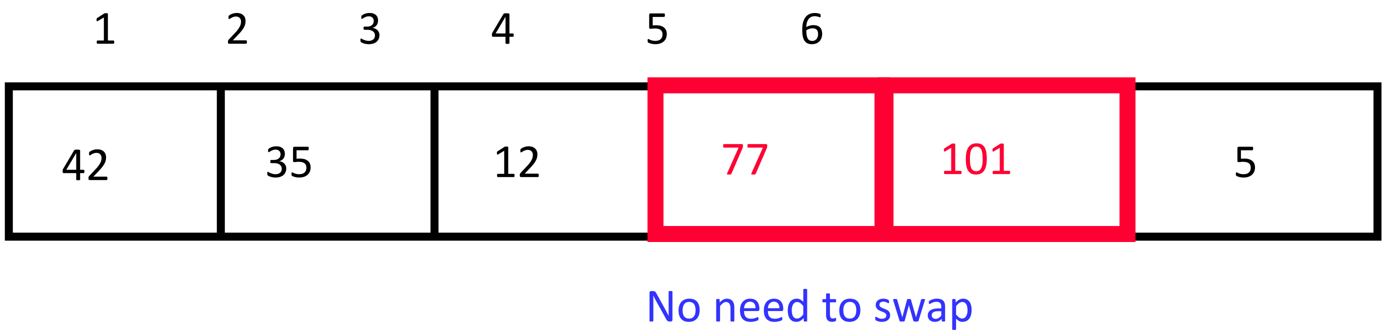
"Bubbling Up" the Largest Element

- **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**



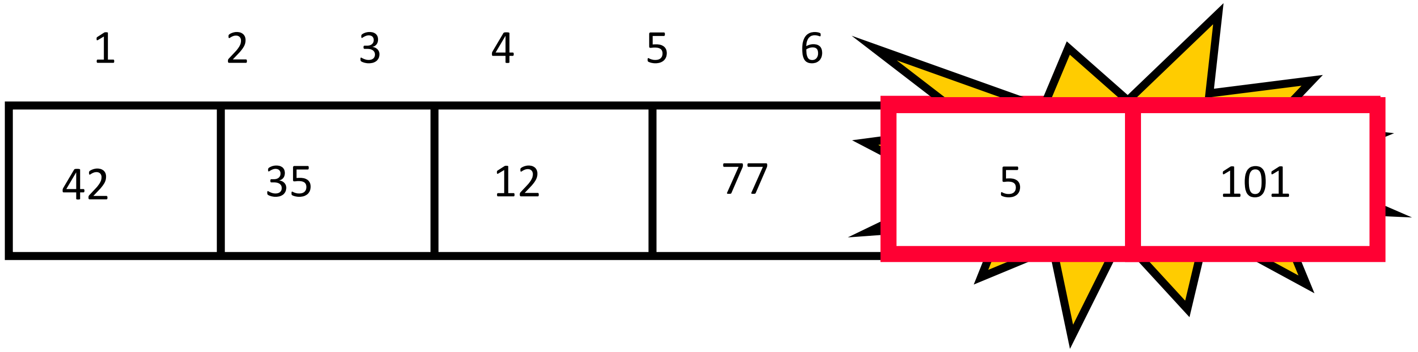
"Bubbling Up" the Largest Element

- **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**



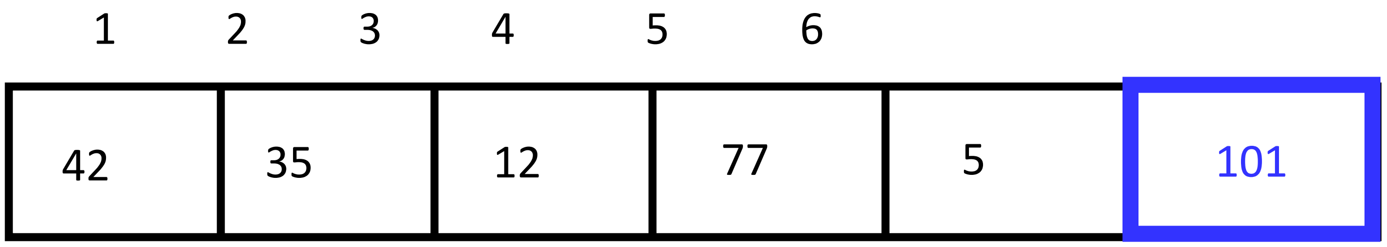
"Bubbling Up" the Largest Element

- **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

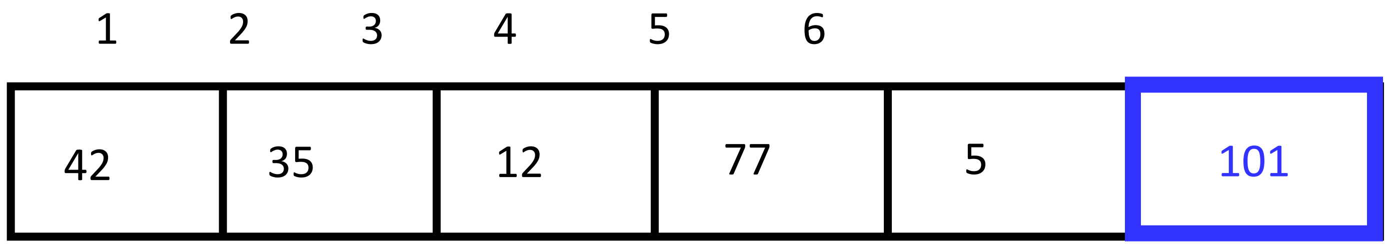
- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



Largest value correctly placed

Items of Interest

- **Notice that only the largest value is correctly placed**
- **All other values are still out of order**
- **So we need to repeat this process**

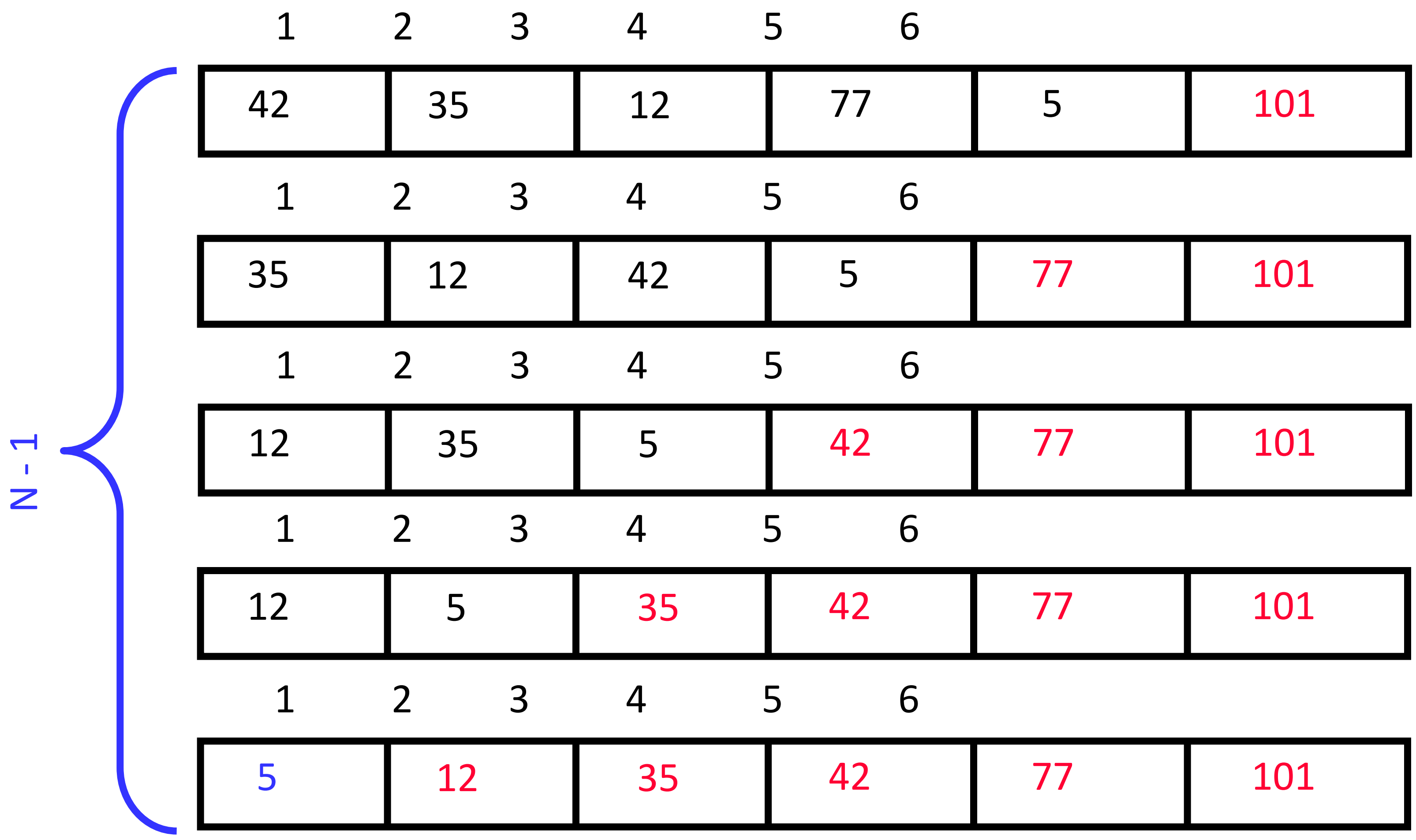


Largest value correctly placed

Repeat “Bubble Up” How Many Times?

- **If we have N elements…**
- **And if each time we bubble an element, we place it in its correct location…**
- **Then we repeat the “bubble up” process N – 1 times.**
- **This guarantees we’ll correctly place all N elements.**

“Bubbling” All the Elements



Algorithm for sorting an array (Bubble sort)

(Bubble sort) BUBBLE (DATA, N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

```
1. Repeat Steps 2 and 3 for K = 1 to N – 1.
2.          Set PTR := 1. **[Initialize pass pointer PTR.]**
3.          Repeat while PTR <= (N – K): **[Executes pass.]**
                 (a) If DATA[PTR] > DATA[PTR + 1], then:
                          Interchange DATA[PTR] and DATA[PTR + 1].
                     **[End of If structure.]**
                 (b) Set PTR := PTR + 1.
            **[End of inner loop.]**
       **[End of Step 1 outer loop.]**
4. Exit.
```

Summary

- "Bubble Up" algorithm will move largest value to its correct location (to the right)
- Repeat "Bubble Up" until all elements are correctly placed:
  - Maximum of N-1 times
  - Can finish early if no swapping occurs
- We reduce the number of elements we compare each time one is correctly placed

Selection Sort

Selection Sort

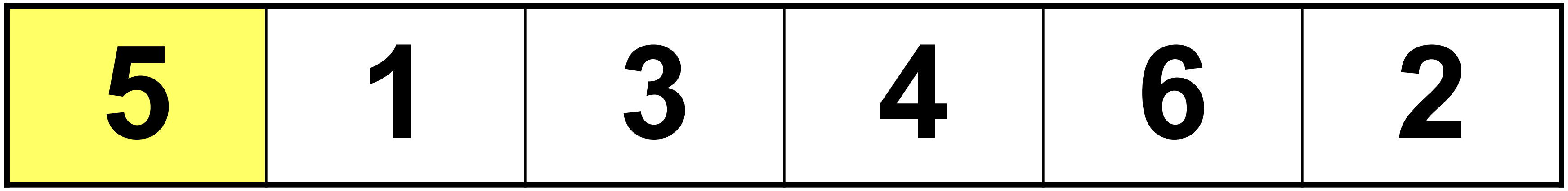
- Selection sort is a simple sorting algorithm. This sorting algorithm is a in-place comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end.
- Smallest element is selected from the unsorted array and swapped with the leftmost element
- This process continues moving unsorted array boundary by one element to the right.

Selection Sort

5	**1**	**3**	**4**	**6**	**2**

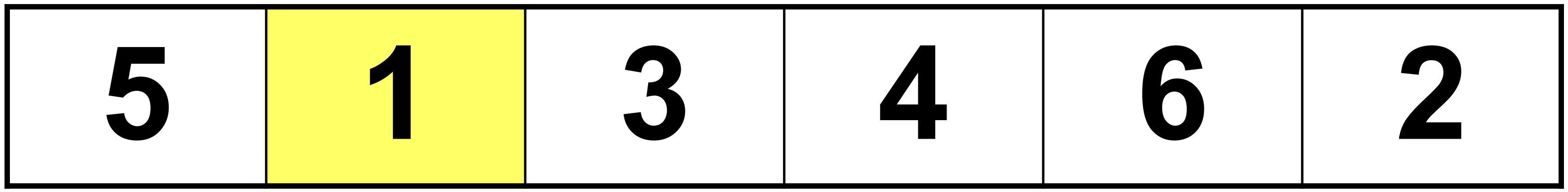
🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort



🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort

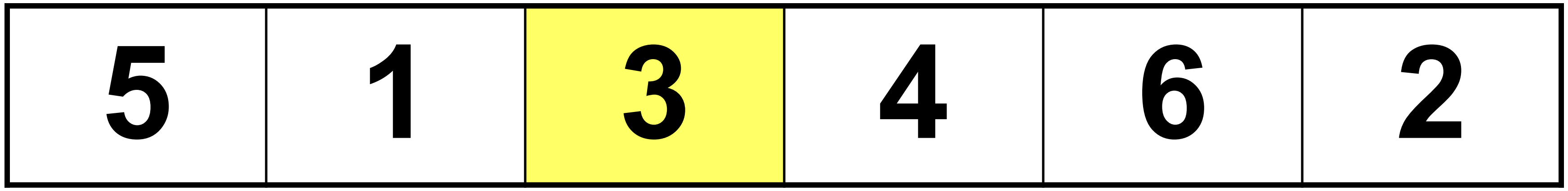


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

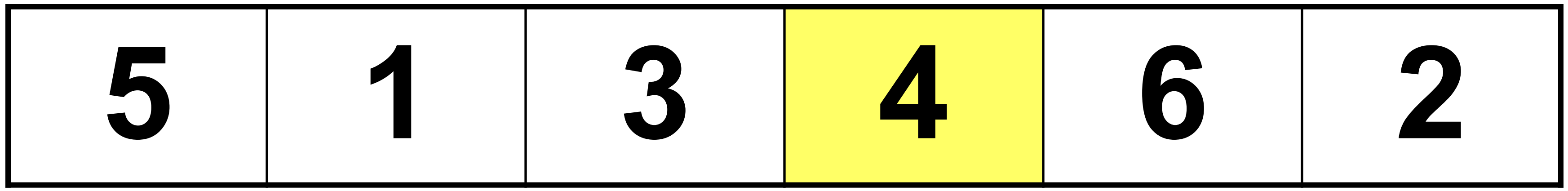


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

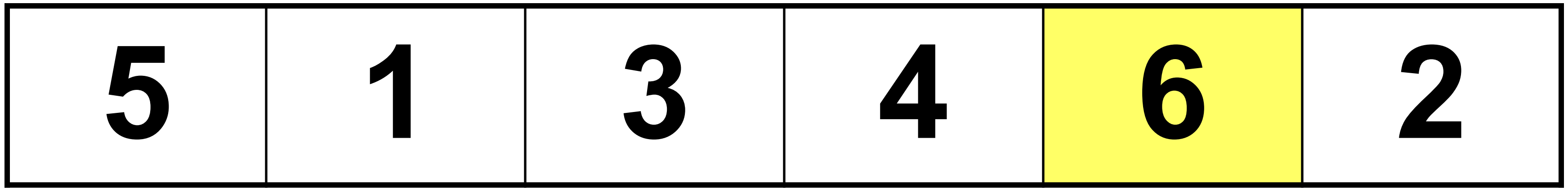


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

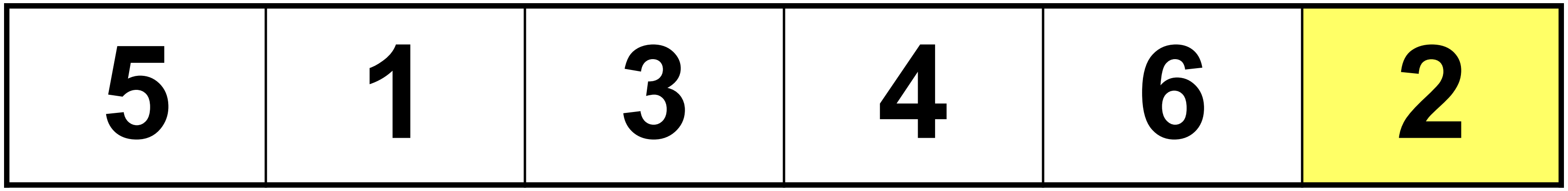


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



🟨	Comparison
🟩	Data Movement
🟦	Sorted

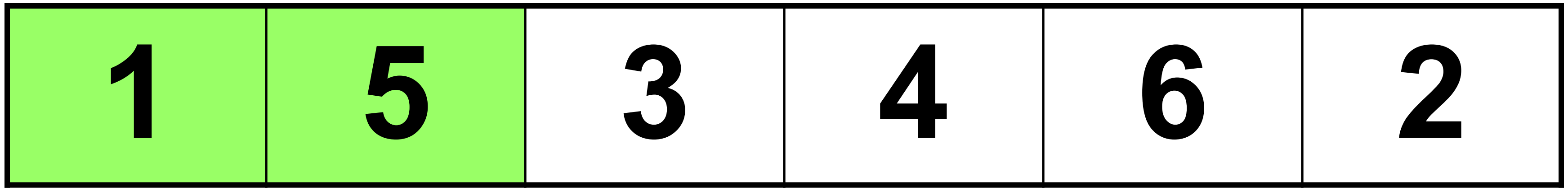
Selection Sort

5	**1**	**3**	**4**	**6**	**2**

	□ **Min**				

🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort



🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

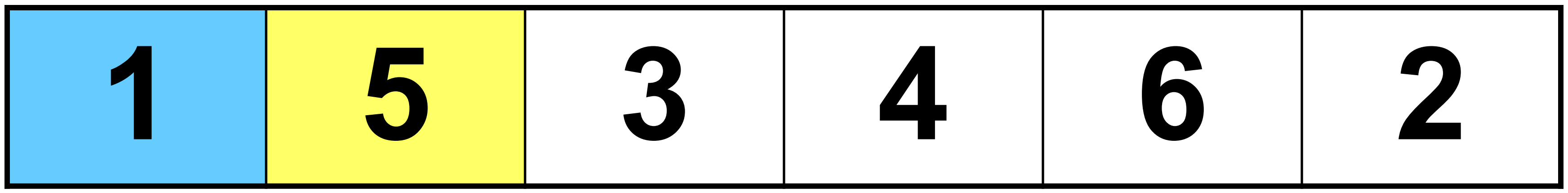
1	5	3	4	6	🟦 2

🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

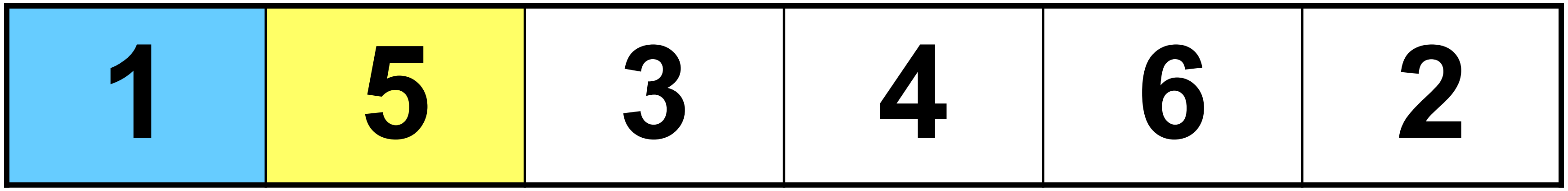


🟨 Comparison

🟩 Data Movement

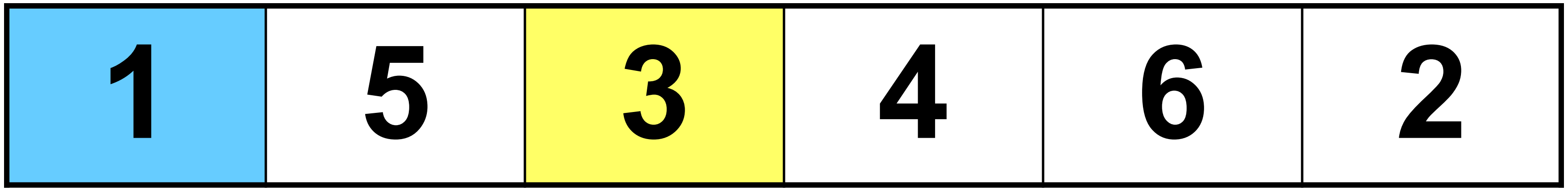
🟦 Sorted

Selection Sort



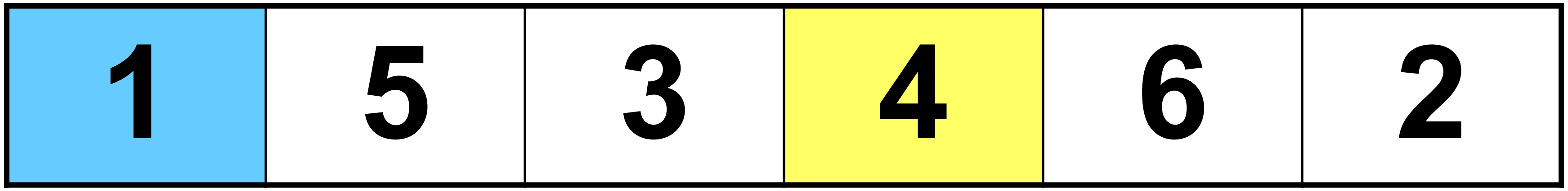
🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort



🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort

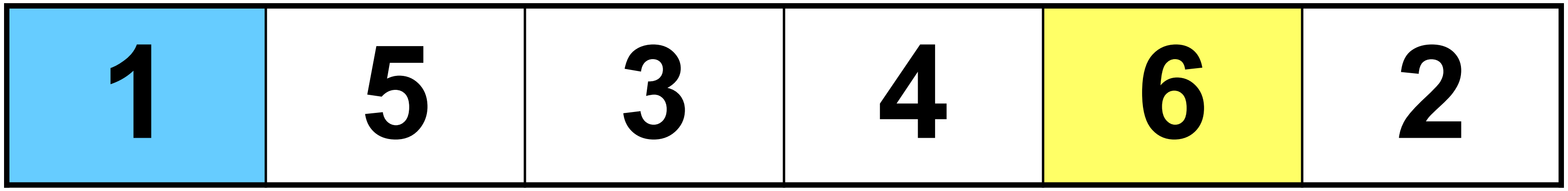


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

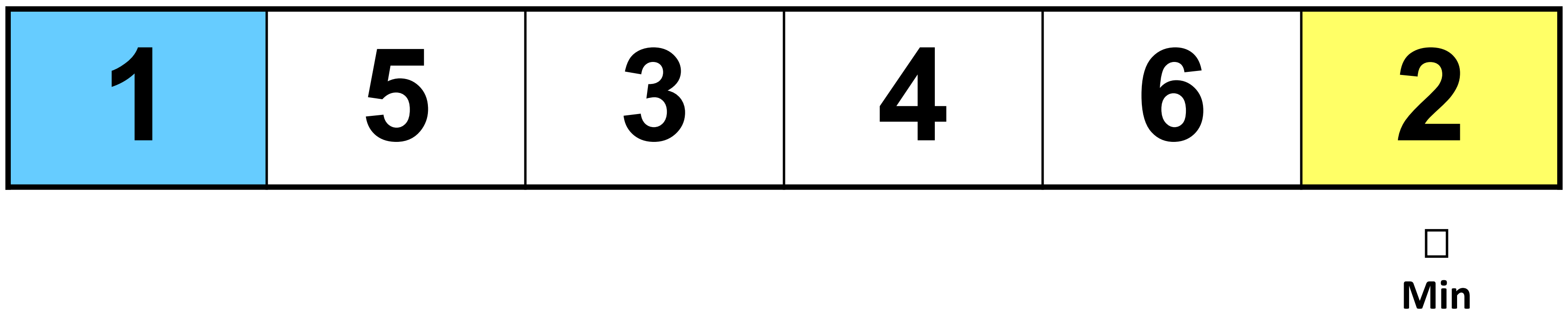


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

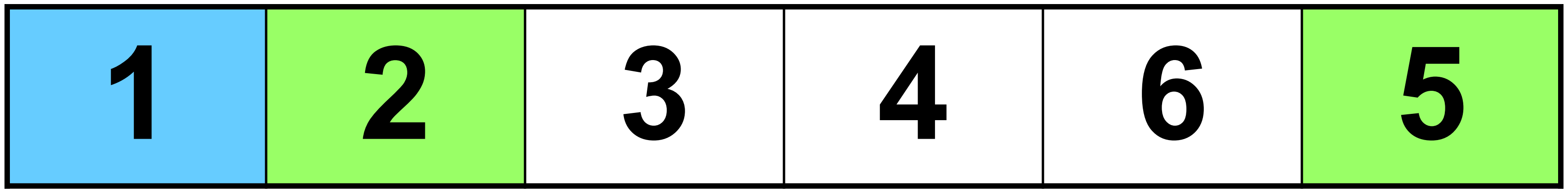


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

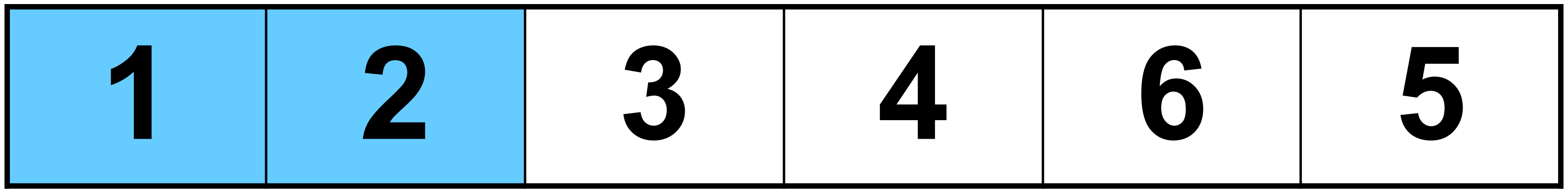


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

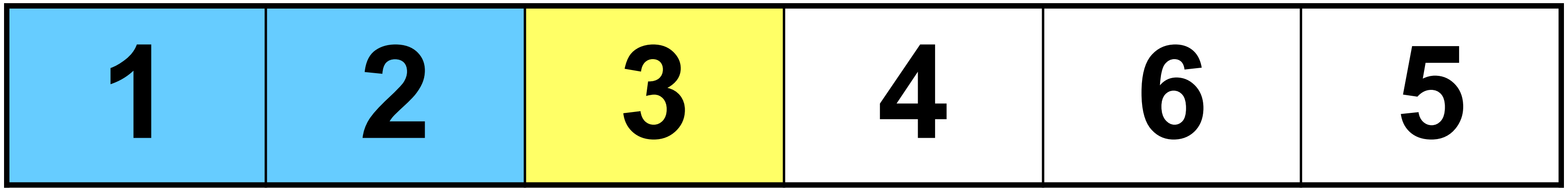


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

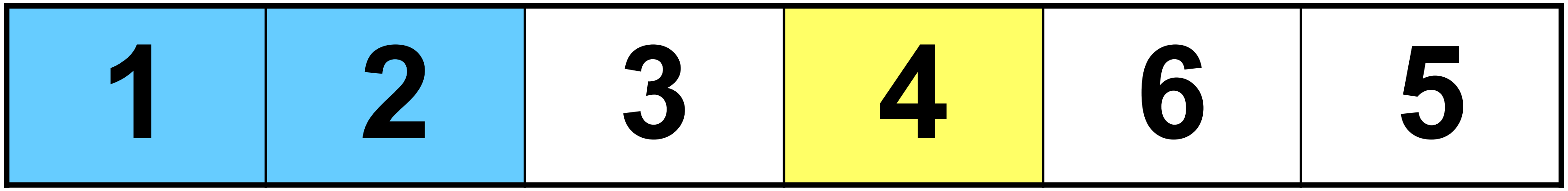


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

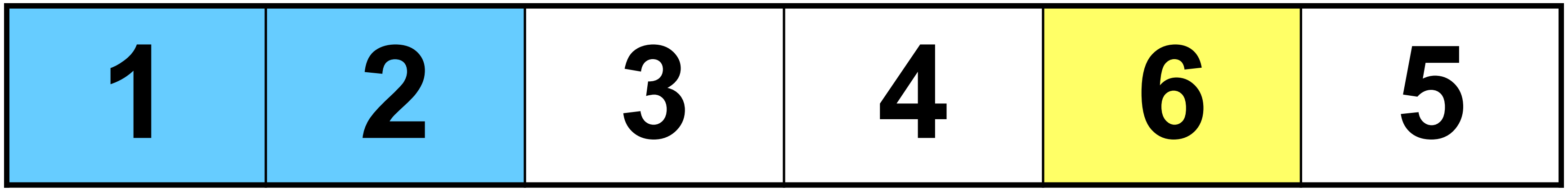


🟨 Comparison

🟩 Data Movement

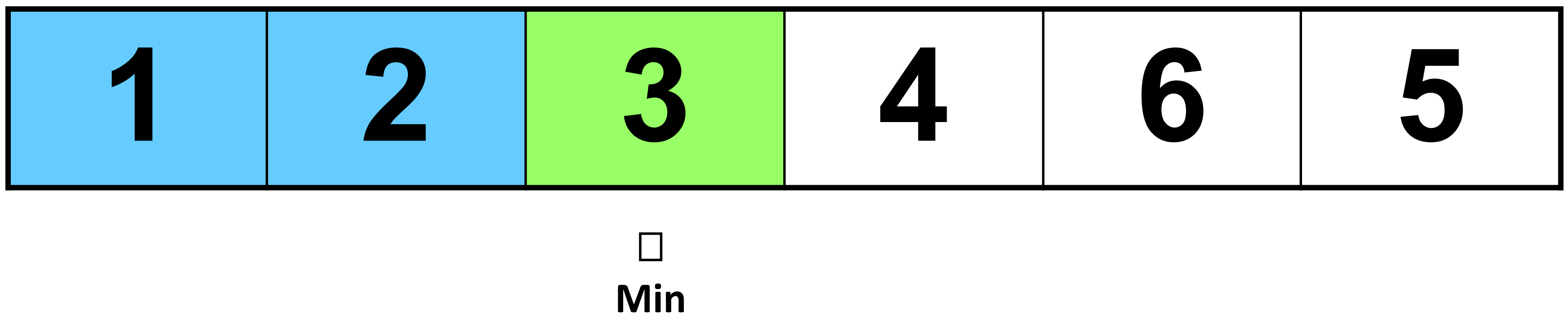
🟦 Sorted

Selection Sort



🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort

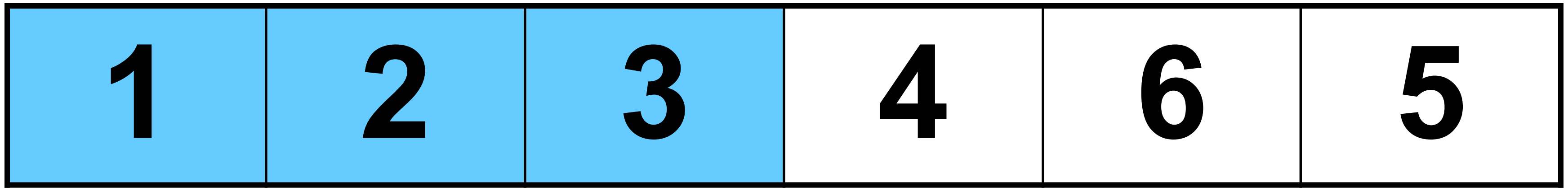


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

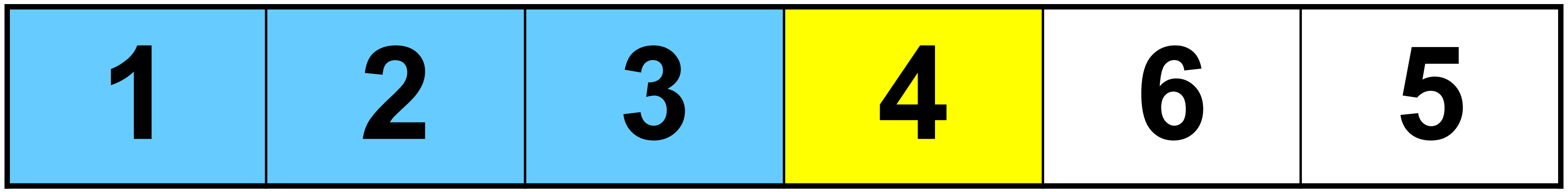


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

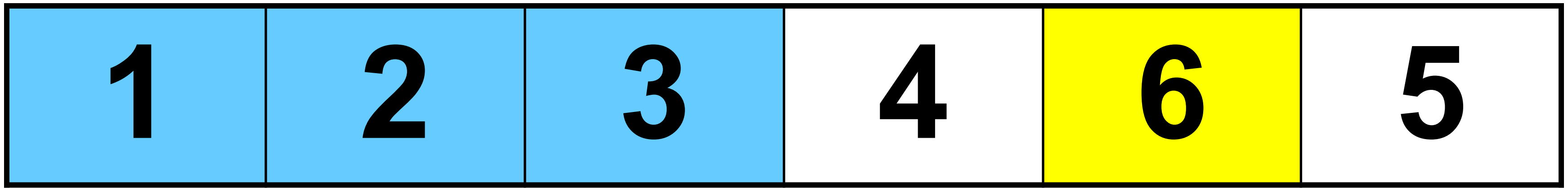


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

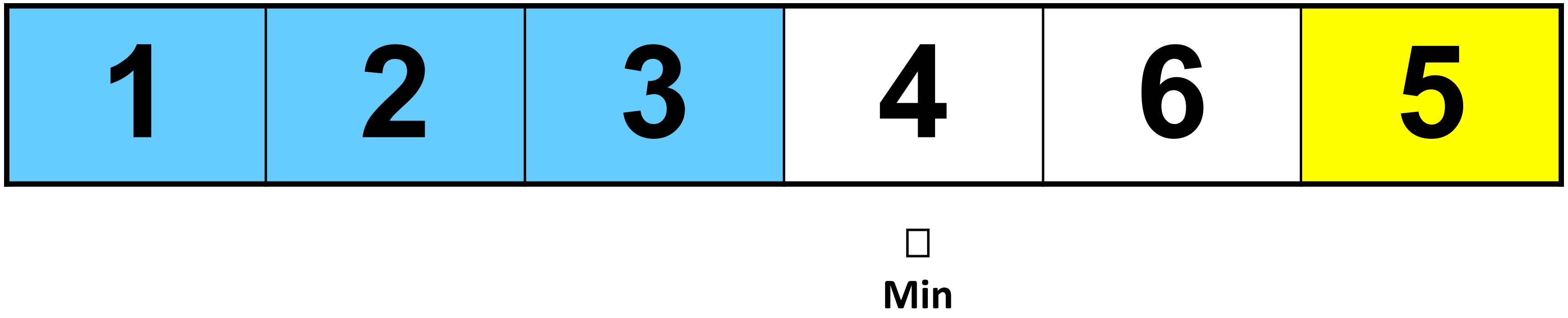


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

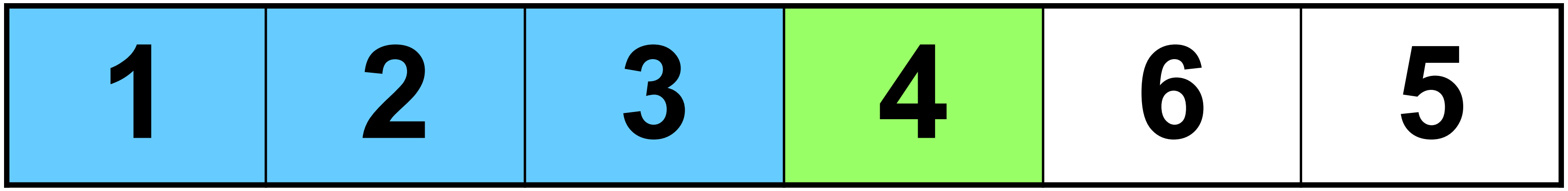


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

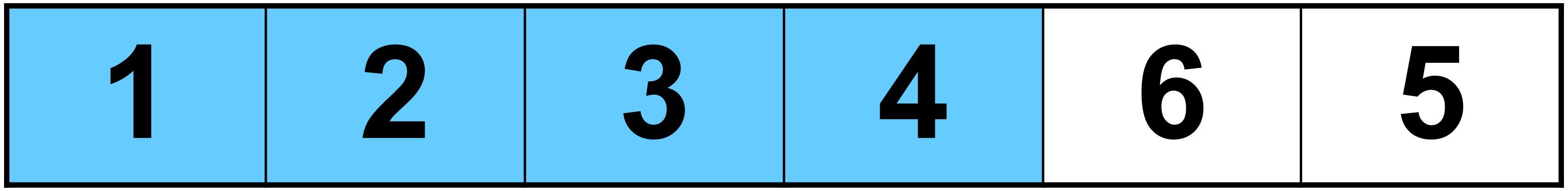


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

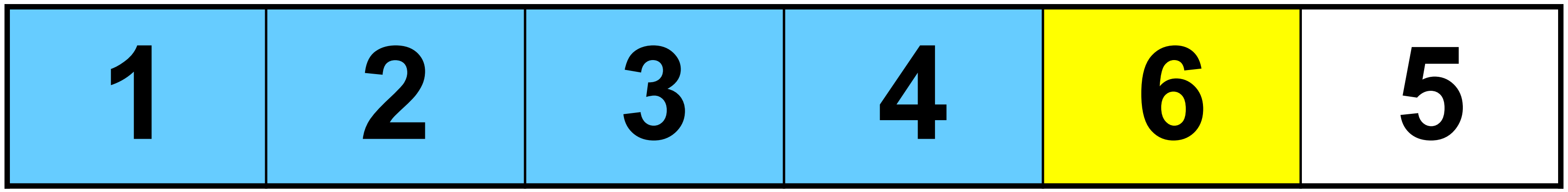


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

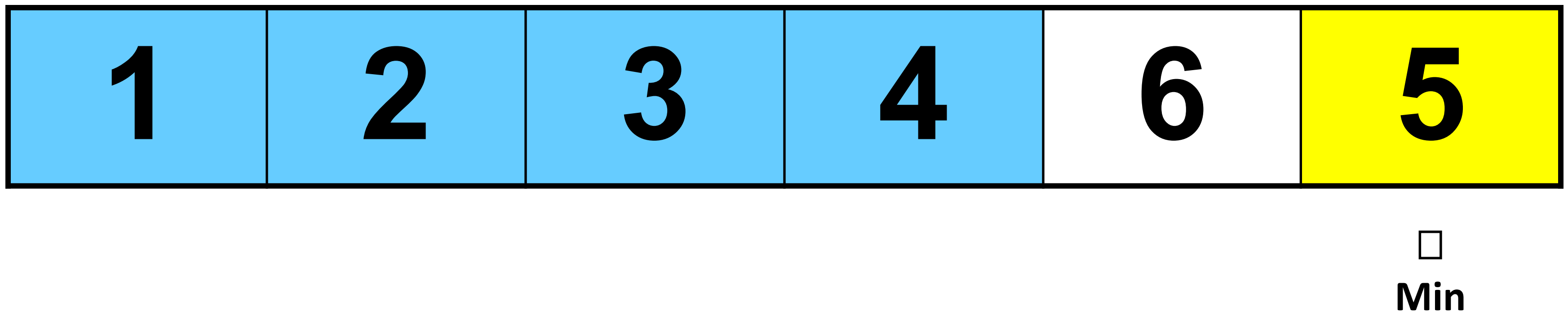


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

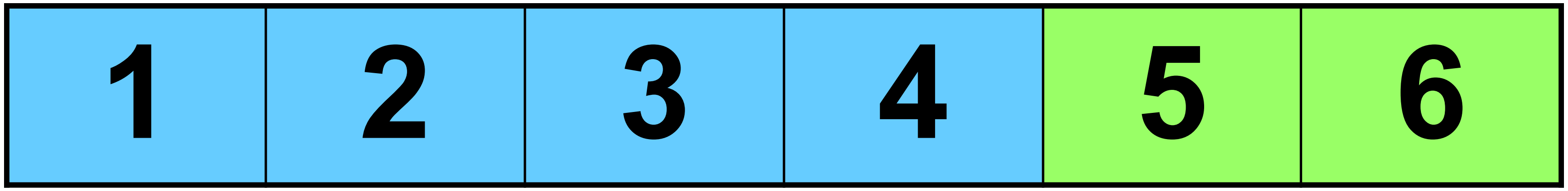


🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort



🟨 Comparison

🟩 Data Movement

🟦 Sorted

Selection Sort

1	**2**	**3**	**4**	**5**	**6**

DONE!

🟨	Comparison
🟩	Data Movement
🟦	Sorted

Selection Sort

```
Selection_Sort (A, n)
1.    Repeat for k= 1 to n-1
2.   Call Min(A, n, k, loc)
3.   Interchange A[k] with A[loc]
        a)set temp = A[K]
        b)A[k]= A[loc]
        c)A[loc]= temp
End of loop
4.    Exit

Min(A, n, k, loc)
1.    Set min= A[K] & loc= k
2.   Repeat for j= k+1, k+2, ……., n
        If(min>A[j]) then
        Set min=A[j] and loc=j
End of loop
3. Exit
```

Insertion Sort

Insertion Sort

- This is a in-place comparison based sorting algorithm. Here, a sub-list is maintained which is always sorted.
- For example, the lower part of an array is maintained to be sorted.
- A element which is to be inserted in this sorted sub-list, has to find its appropriate place and insert it there. Hence the name **insertion sort**.
- The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list (in the same array).

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 0: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

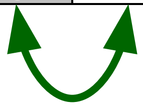
Iteration 1: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	0.56	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71



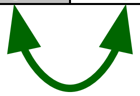
Iteration 2: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71



Iteration 2: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

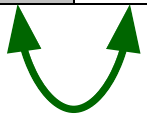
Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.
- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	1.12	7.42	1.17	0.32	6.21	4.42	3.14	7.71



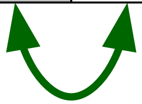
Iteration 3: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71



Iteration 3: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

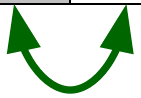
Iteration 3: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	1.17	7.42	0.32	6.21	4.42	3.14	7.71



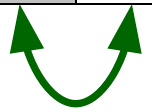
Iteration 4: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71



Iteration 4: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

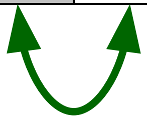
Iteration 4: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	0.32	7.42	6.21	4.42	3.14	7.71



Iteration 5: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	0.32	2.78	7.42	6.21	4.42	3.14	7.71

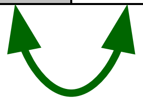
Iteration 5: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	0.32	1.17	2.78	7.42	6.21	4.42	3.14	7.71



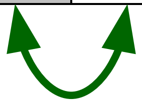
Iteration 5: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	0.32	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71



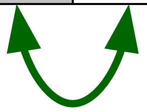
Iteration 5: step 3.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71



Iteration 5: step 4.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

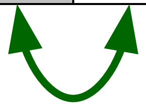
Iteration 5: step 5.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71



Iteration 6: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

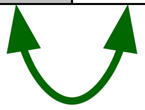
Iteration 6: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	4.42	7.42	3.14	7.71



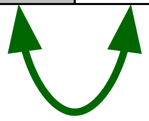
Iteration 7: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71



Iteration 7: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

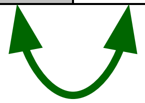
Iteration 7: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	3.14	7.42	7.71



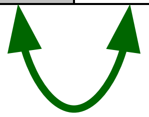
Iteration 8: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	3.14	6.21	7.42	7.71

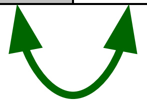


Iteration 8: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.
- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71



Iteration 8: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 3.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 9: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.
- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 10: DONE.

Algorithm

Insertion_Sort (A, n)

```
1.  Repeat for i = 1 to n:
2.      Set j = i
3.      Repeat while j >0 and a[j] < a[j-1]:
4.          Set temp = a[j]
5.          Set a[j] = a[j-1]
6.          Set a[j-1] = temp
7.          Set j = j-1
        [End of step 3 loop.]
8.  [End of step 1 loop.]
9.  Return.
```

Merge Sort

Merge Sort

- Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.

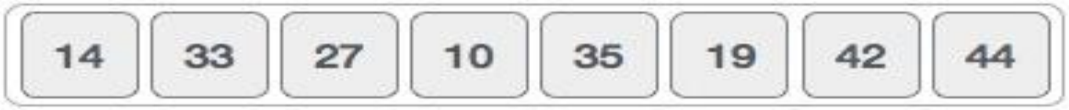
An Example: Merge Sort

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

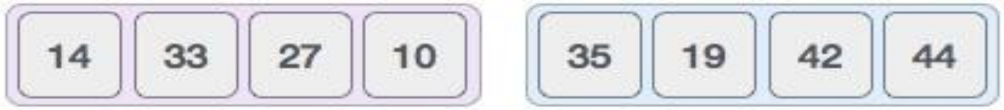
- ***Divide***: Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

Merge Sort

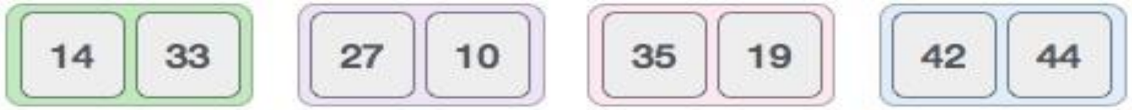
- To understand merge sort, we take an unsorted array as depicted below −



- We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

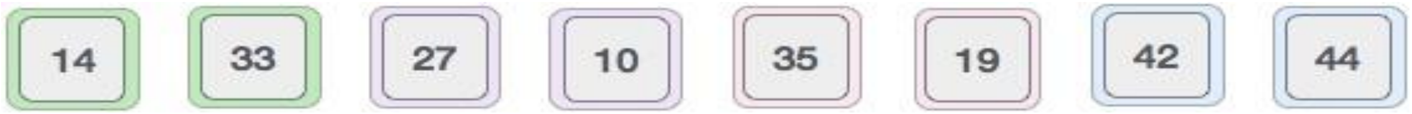


- This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

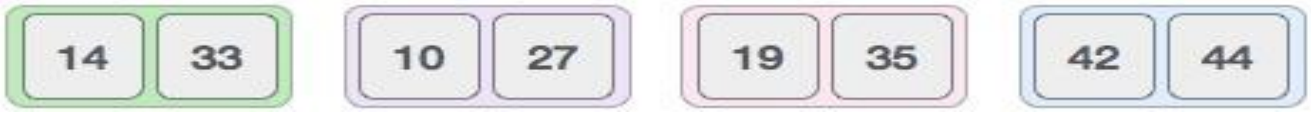


Merge Sort

- We further divide these arrays and we achieve atomic value which can no more be divided.

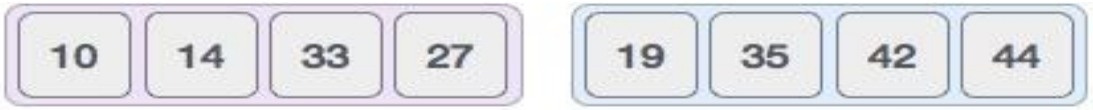


- Now, we combine them in exactly same manner they were broken down. Please note the color codes given to these lists.
- We first compare the element for each list and then combine them into another list in sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order 19 and 35. 42 and 44 are placed sequentially.

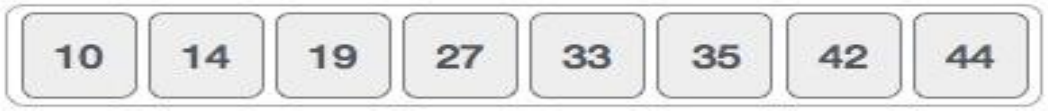


Merge Sort

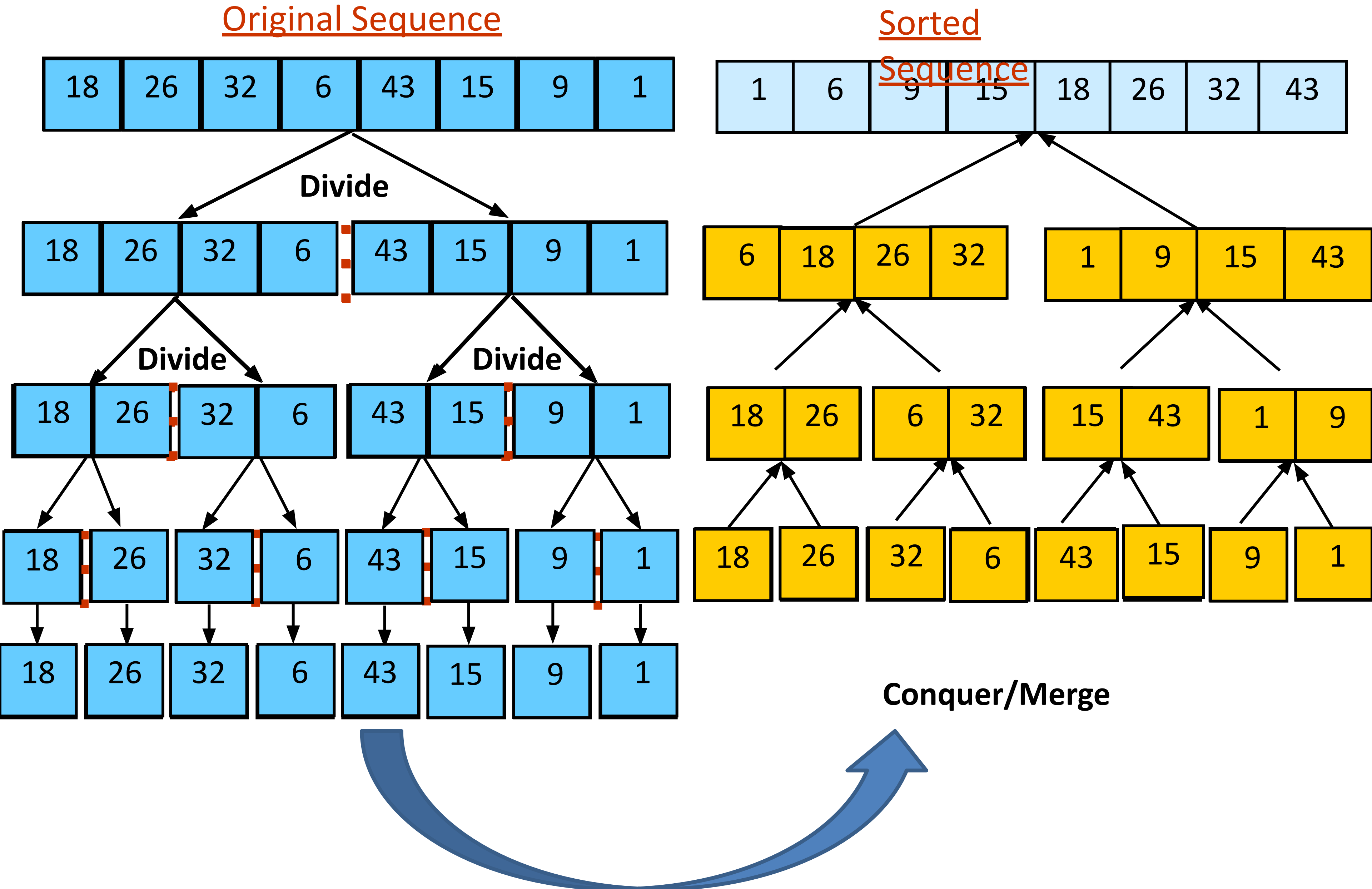
- In next iteration of combining phase, we compare lists of two data values, and merge them into a list of foud data values placing all in sorted order.



- After final merging, the list should look like this −



Merge Sort – Example



Merge-Sort (A, fst, lst)

INPUT: a sequence of n numbers stored in array A

OUTPUT: an ordered sequence of n numbers

```
MergeSort (A, fst, lst)   // sort A[fst...lst] by divide & conquer
1    if fst< lst
2       then mid ← ⌊(fst+lst)/2⌋
3            MergeSort (A, fst, mid)
4            MergeSort (A, mid+1, lst)
5            Merge (A, fst, mid, lst)        // merges   A[fst..mid]   with  A[mid+1..lst]
```

Initial Call: MergeSort(A, 1, n)

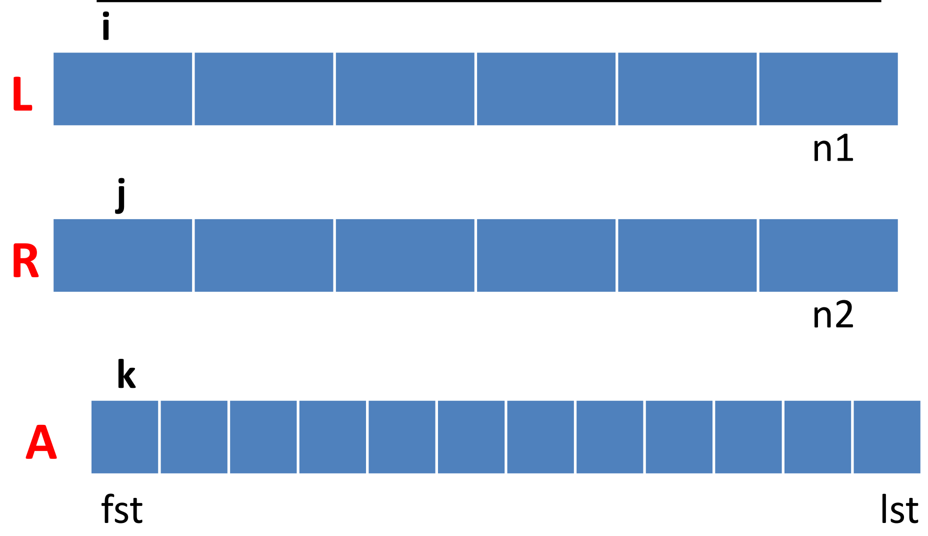
Procedure Merge

```
Merge(A, fst, mid, lst)
$n_1 \leftarrow (mid - fst + 1)$
$n_2 \leftarrow (lst - mid)$

    for $i \leftarrow 1$ to $n_1$
        do $L[i] \leftarrow A[fst + i - 1]$
    for $j \leftarrow 1$ to $n_2$
        do $R[j] \leftarrow A[mid + j]$

    $L[n_1+1] \leftarrow \infty$
    $R[n_2+1] \leftarrow \infty$
    $i \leftarrow 1$
    $j \leftarrow 1$
    for $k \leftarrow fst$ to $lst$
        do if $L[i] \le R[j]$
            then $A[k] \leftarrow L[i]$
                 $i \leftarrow i + 1$
            else $A[k] \leftarrow R[j]$
                 $j \leftarrow j + 1$
```

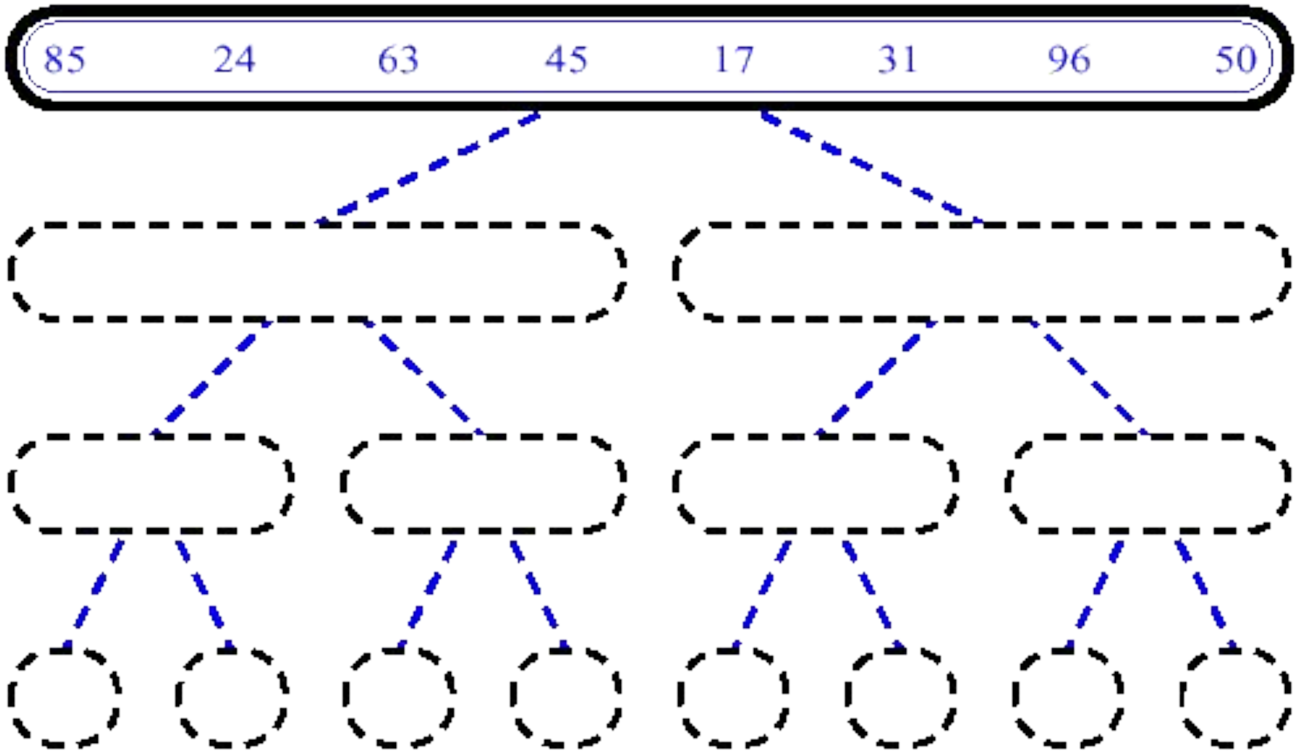
Input: Array containing sorted subarrays $A[fst...mid]$ and $A[mid+1...lst]$

Output: Merged sorted subarray in $A[fst...lst]$.

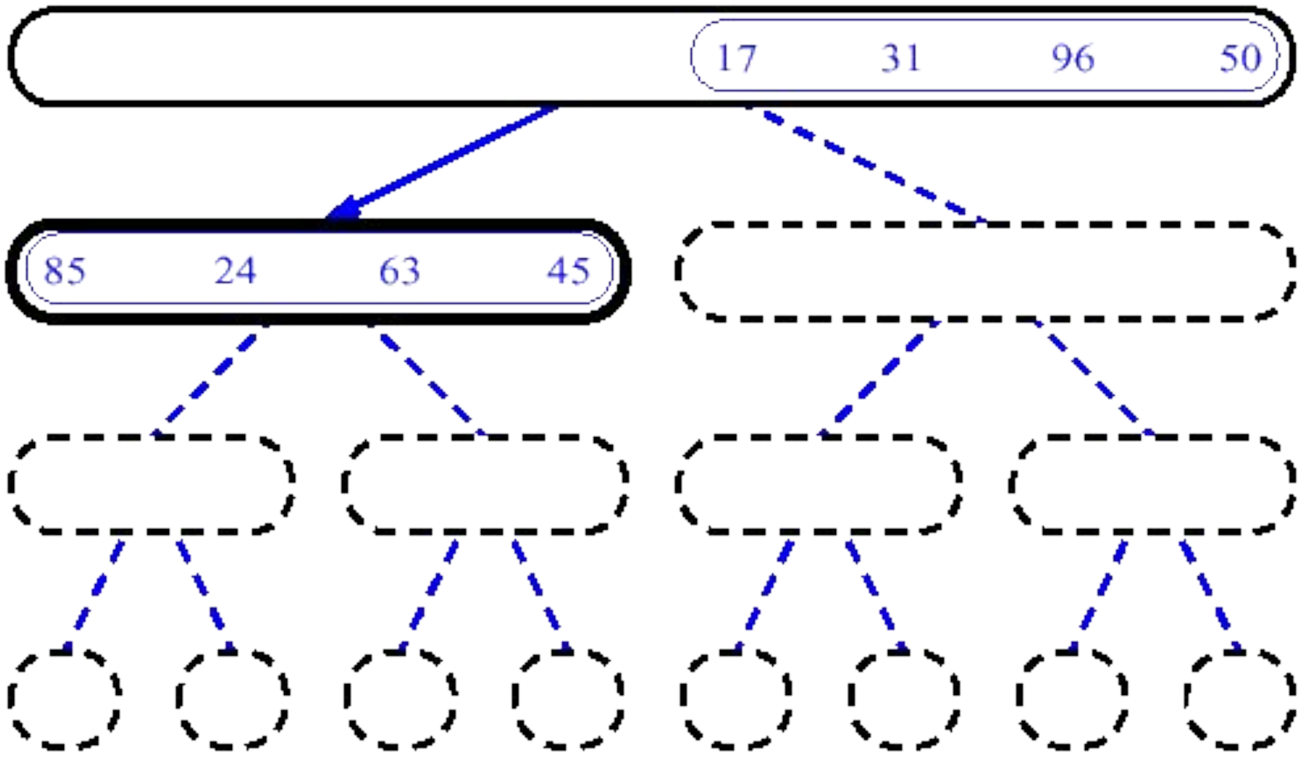
Sentinels, to avoid having to check if either subarray is fully copied at each step.



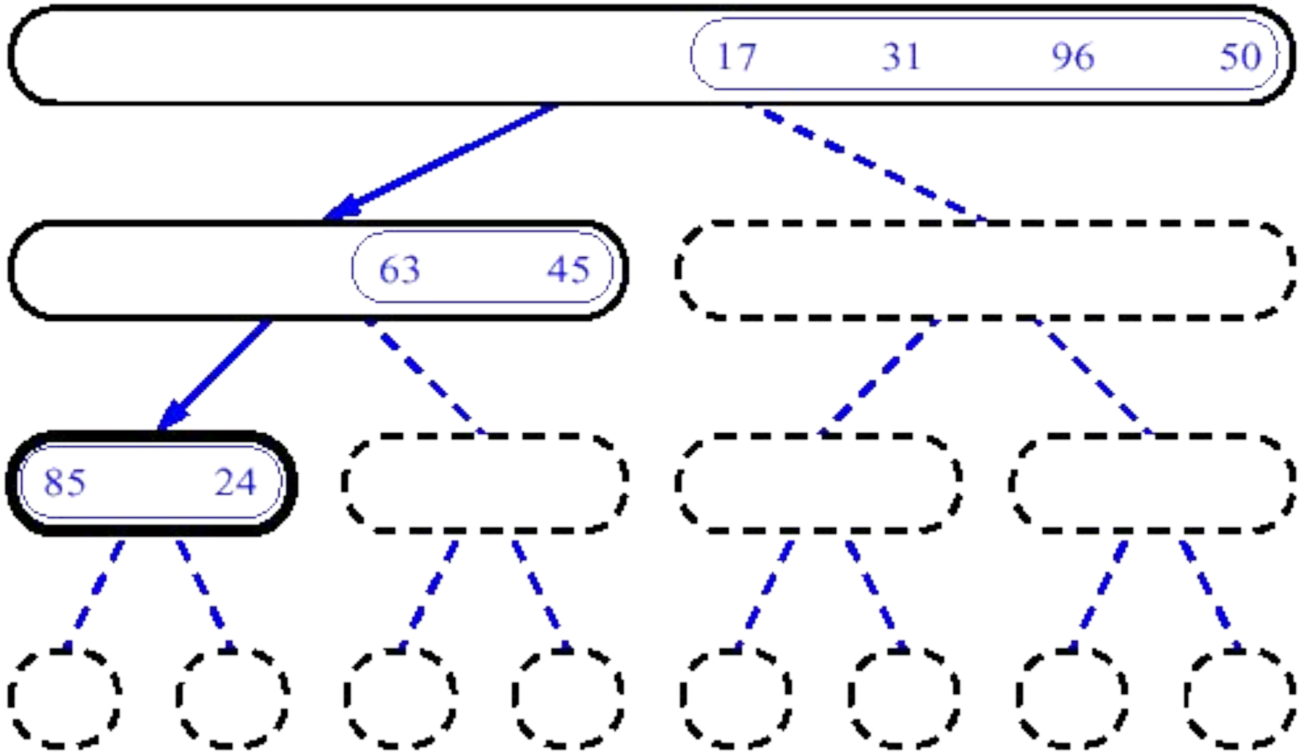
MergeSort (Example) - 1



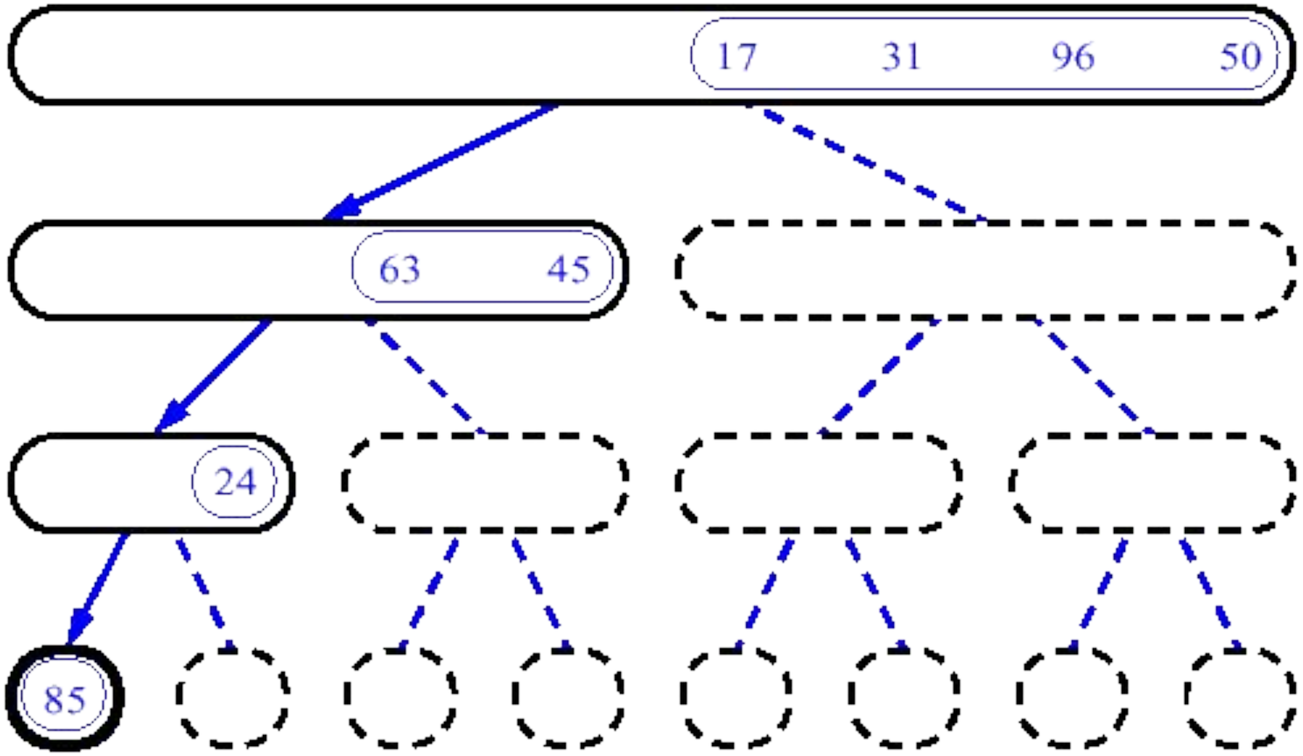
MergeSort (Example) - 2



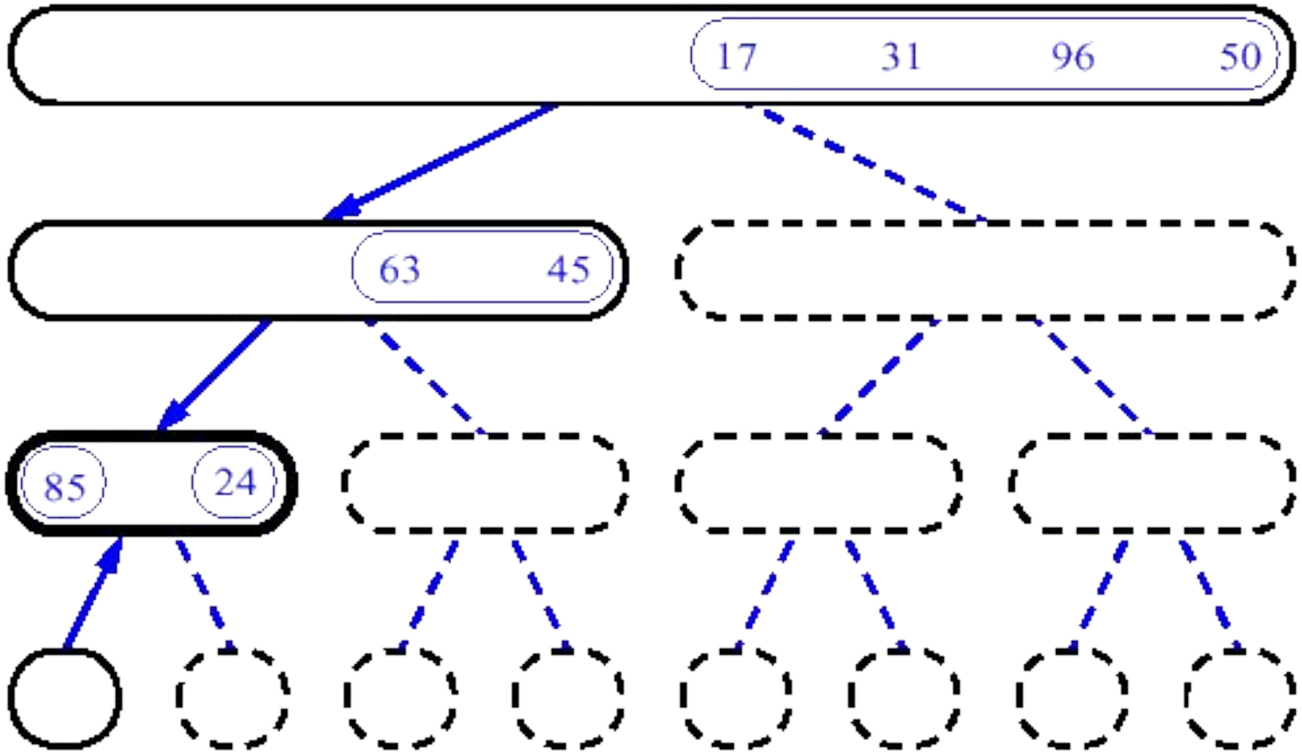
MergeSort (Example) - 3



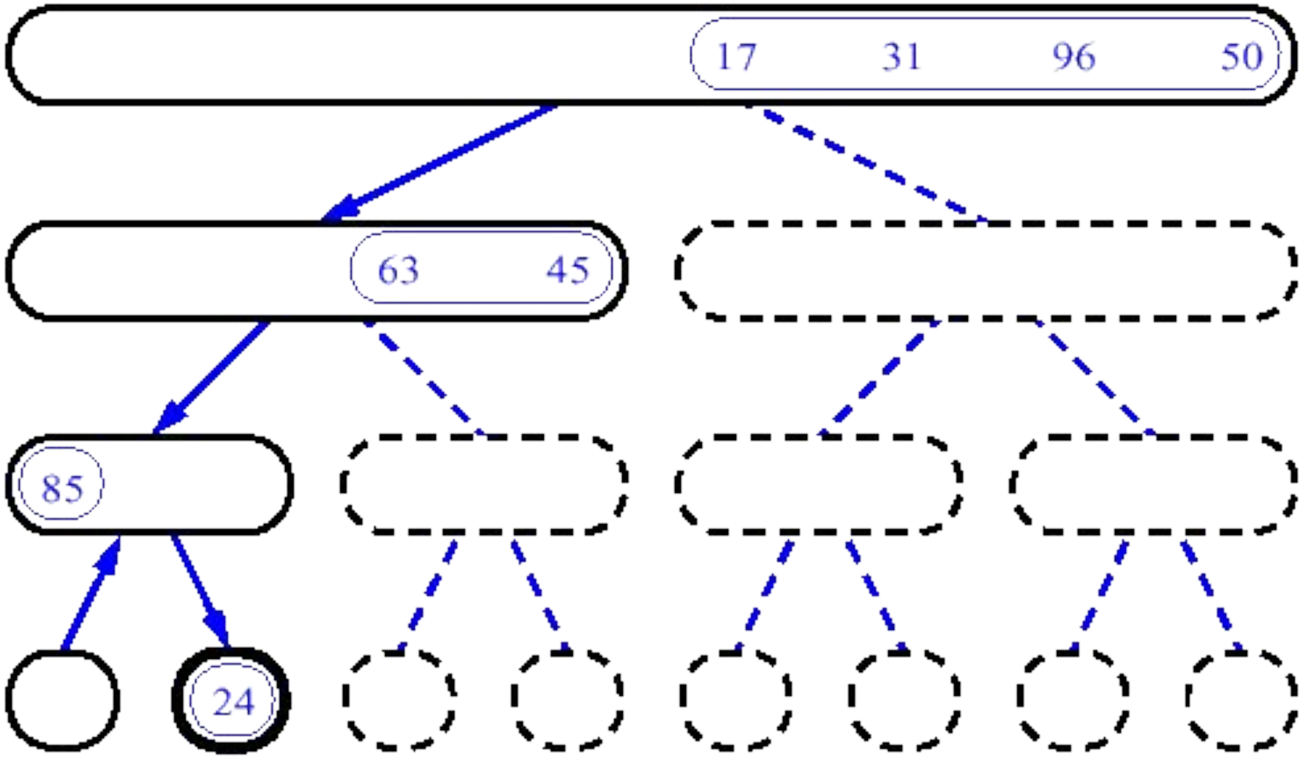
MergeSort (Example) - 4



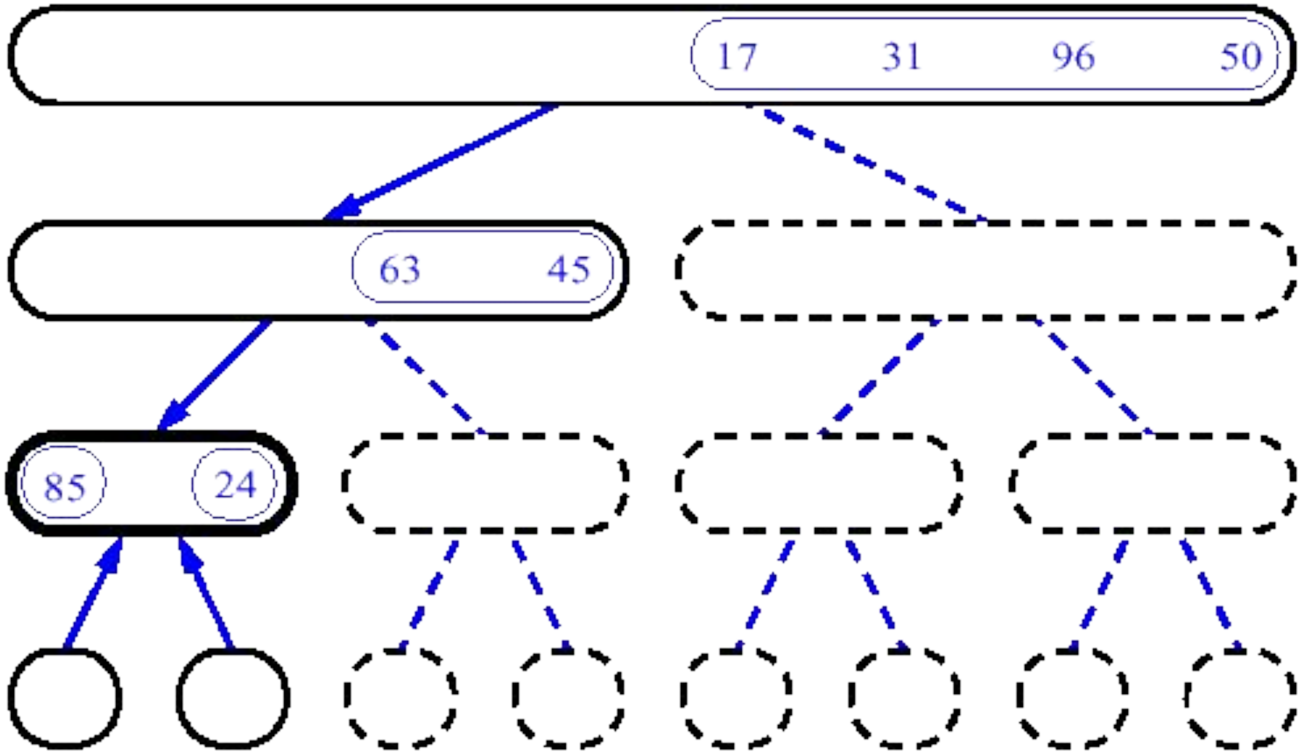
MergeSort (Example) - 5



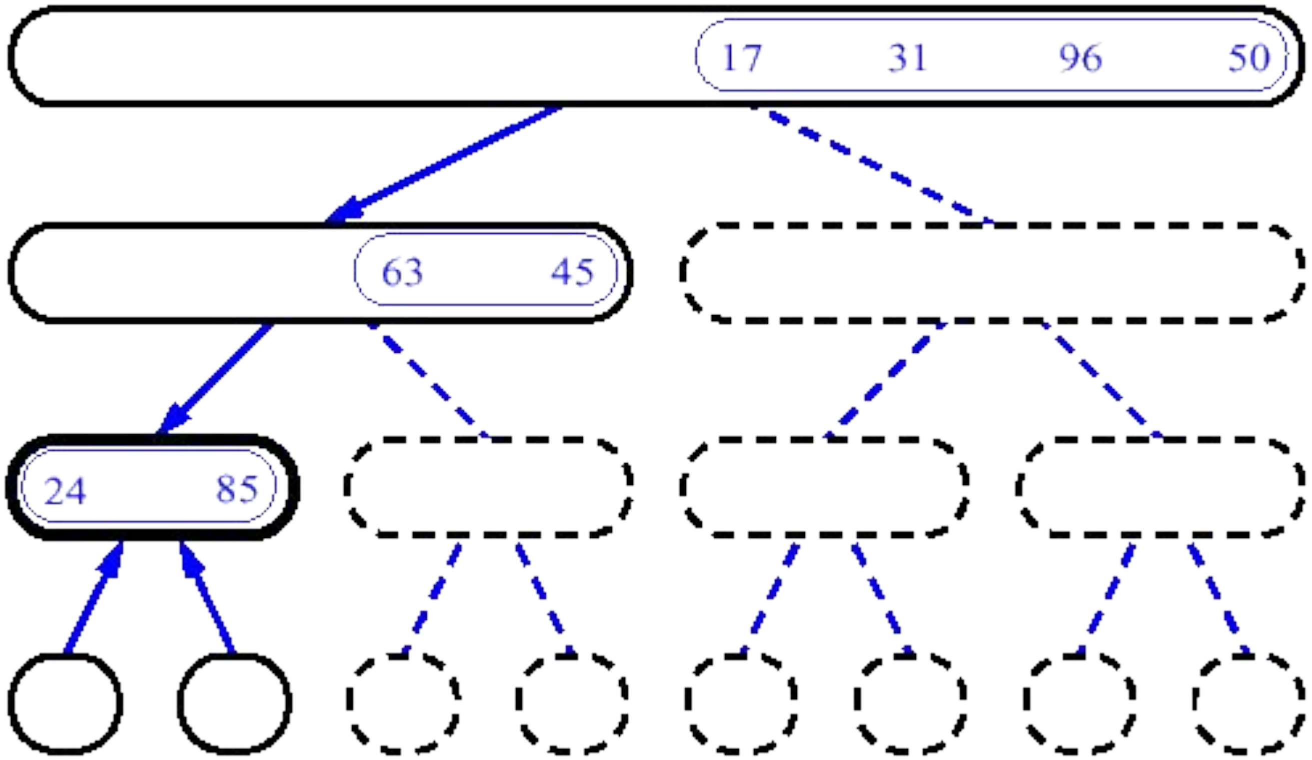
MergeSort (Example) - 6



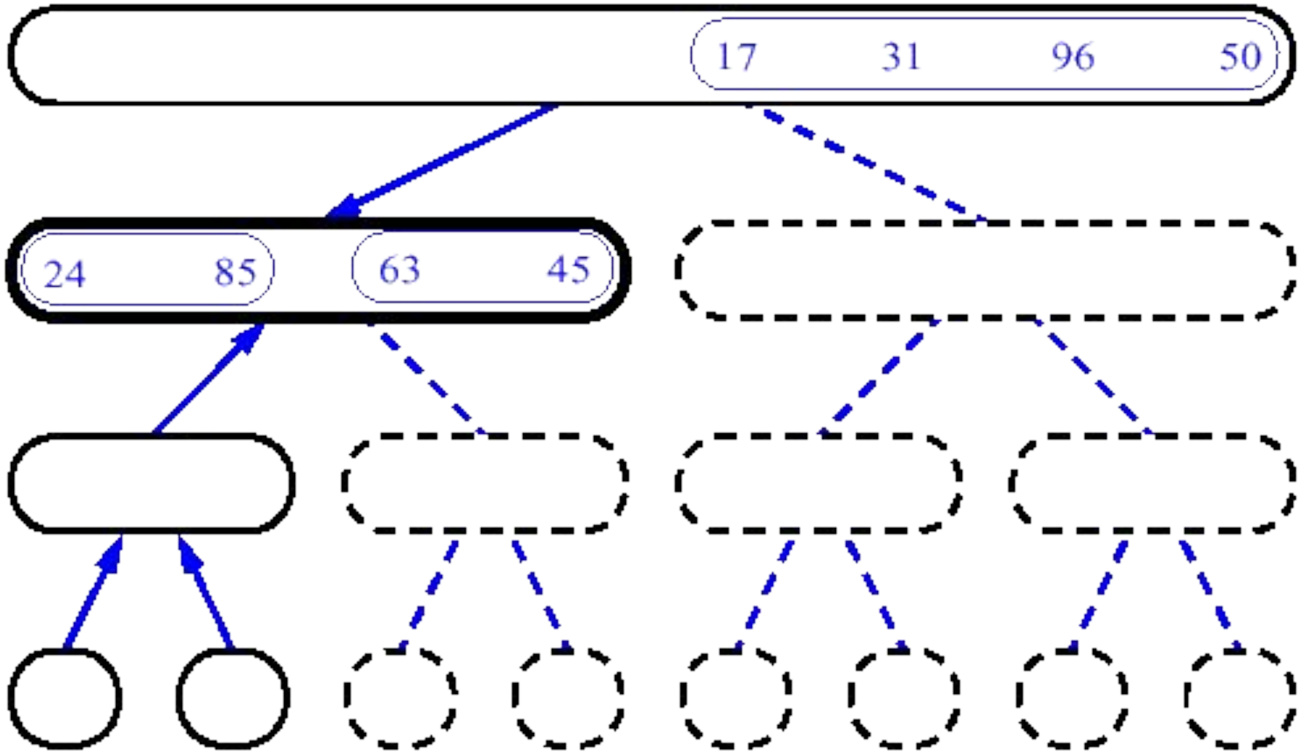
MergeSort (Example) - 7



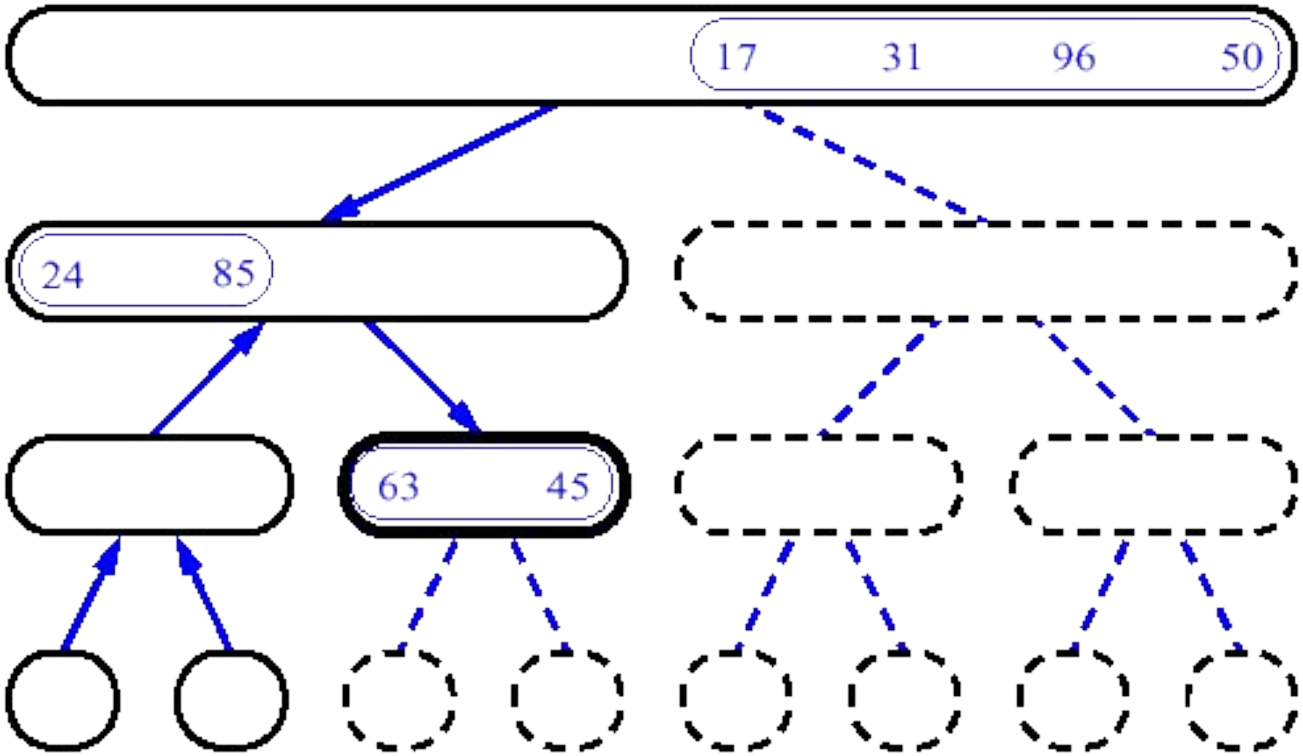
MergeSort (Example) - 8



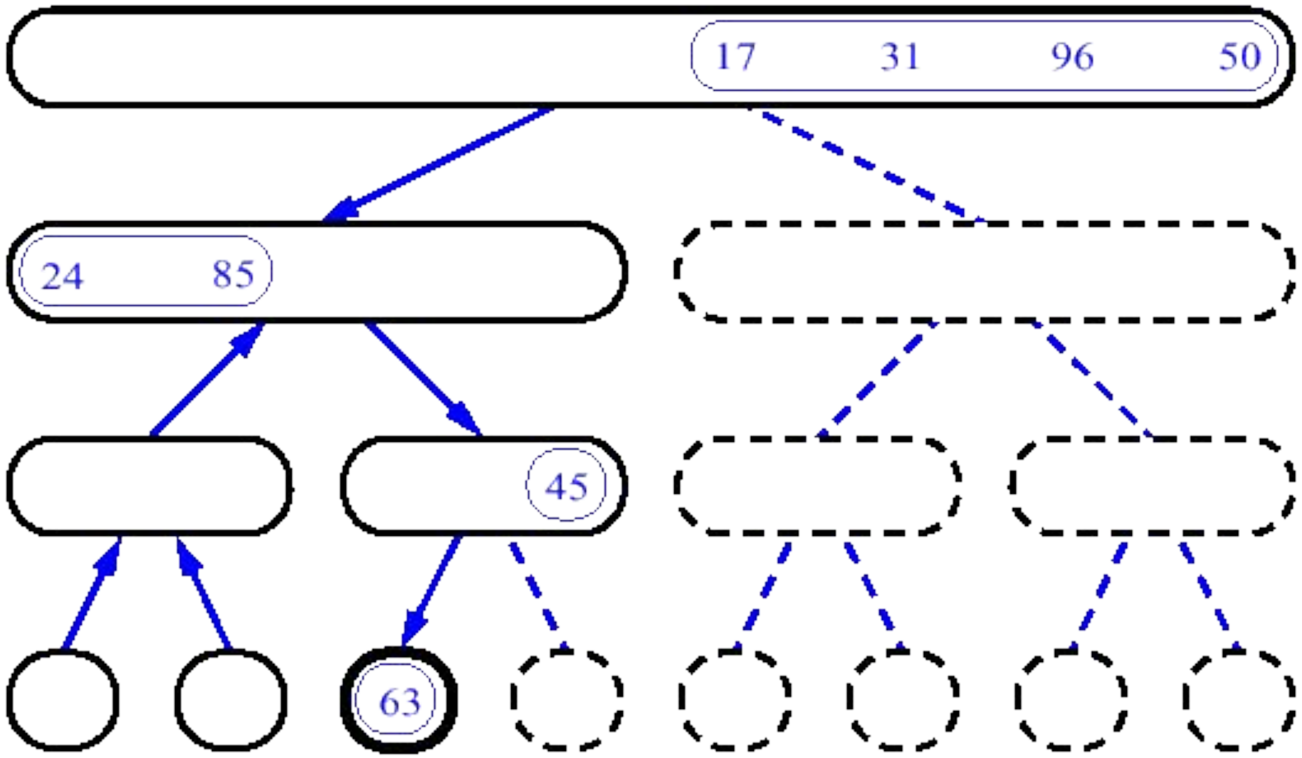
MergeSort (Example) - 9



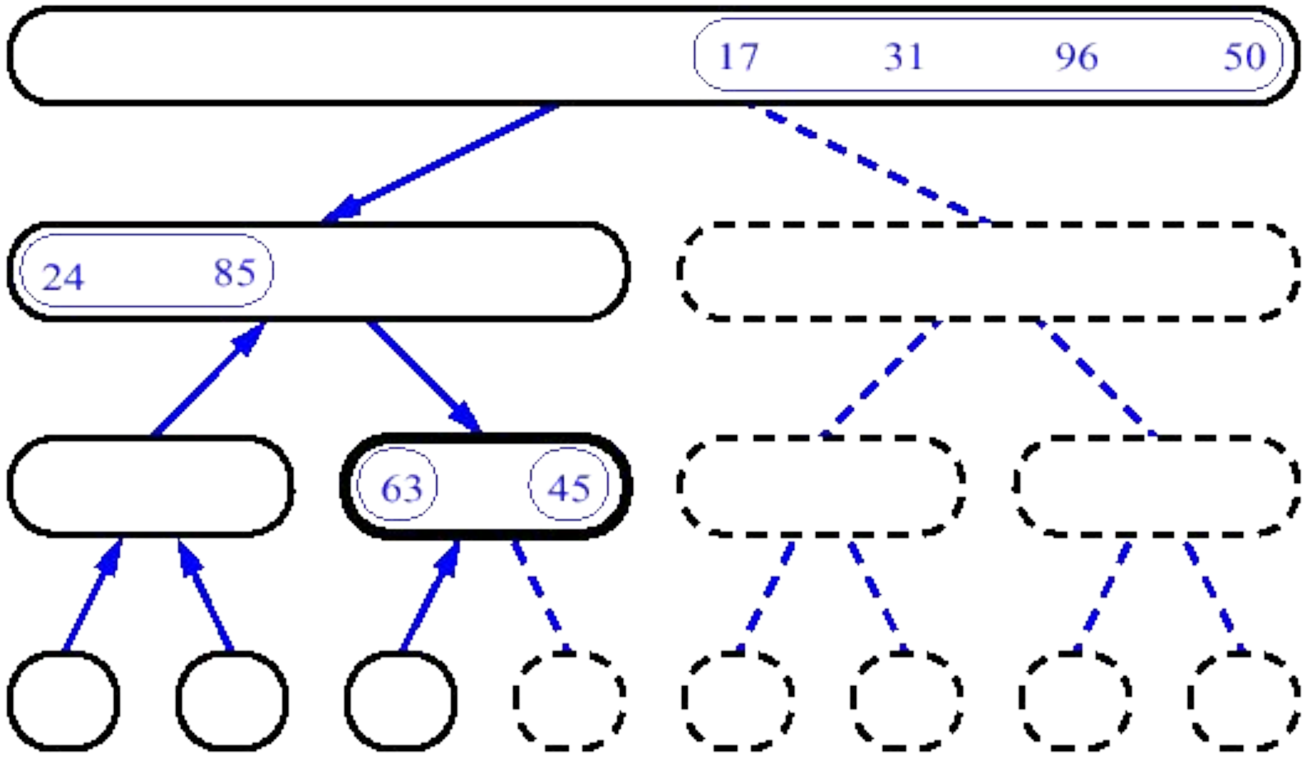
MergeSort (Example) - 10



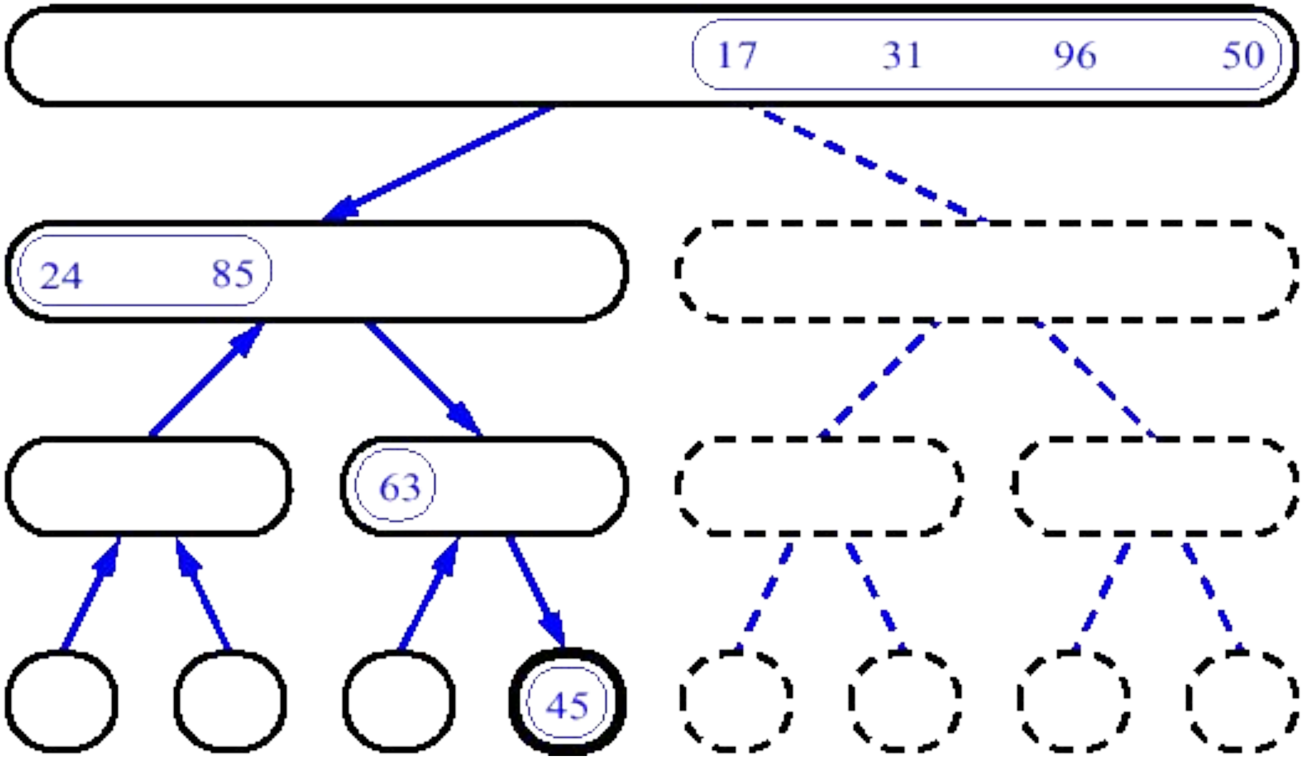
MergeSort (Example) - 11



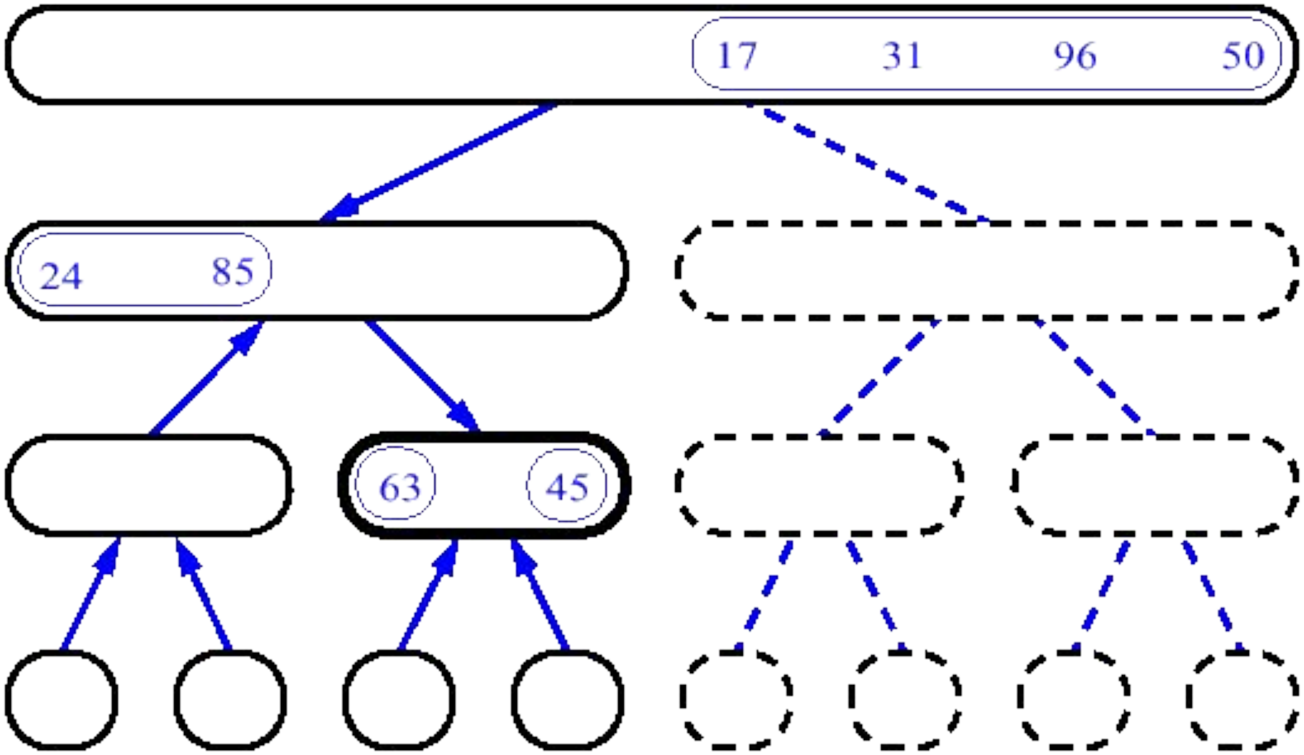
MergeSort (Example) - 12



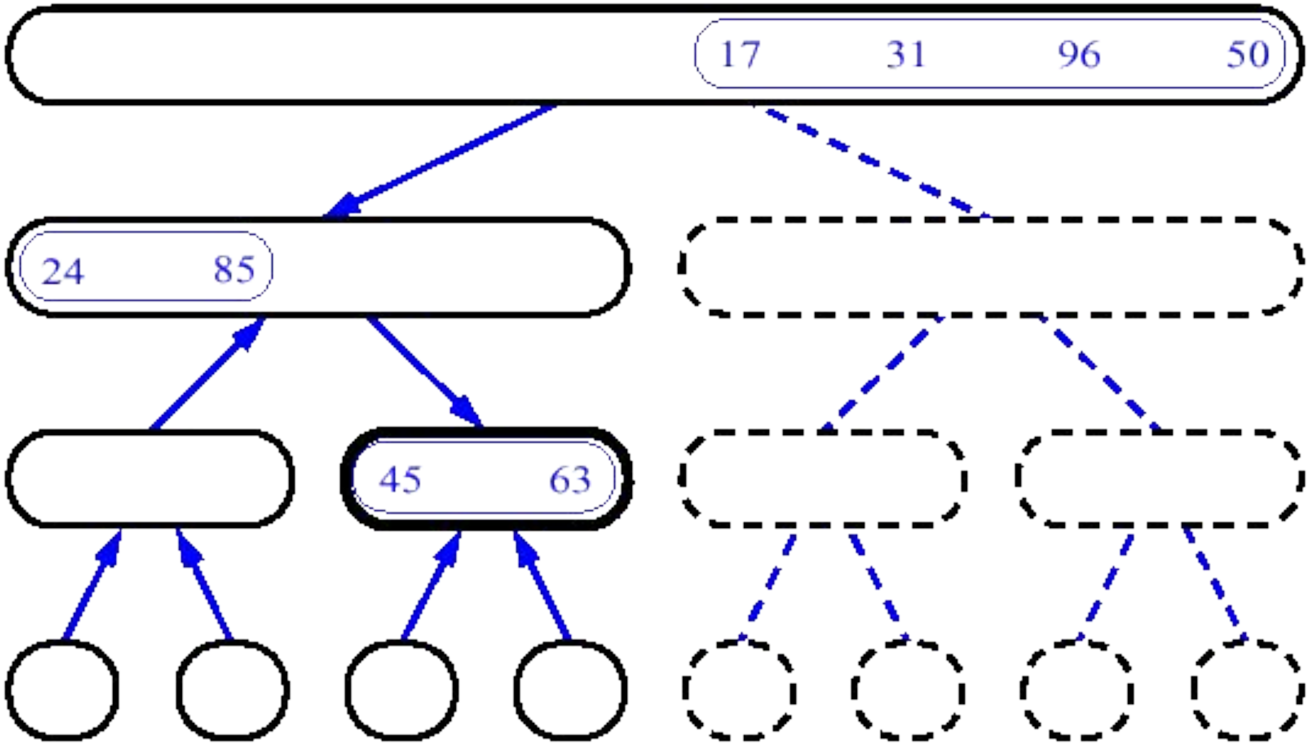
MergeSort (Example) - 13



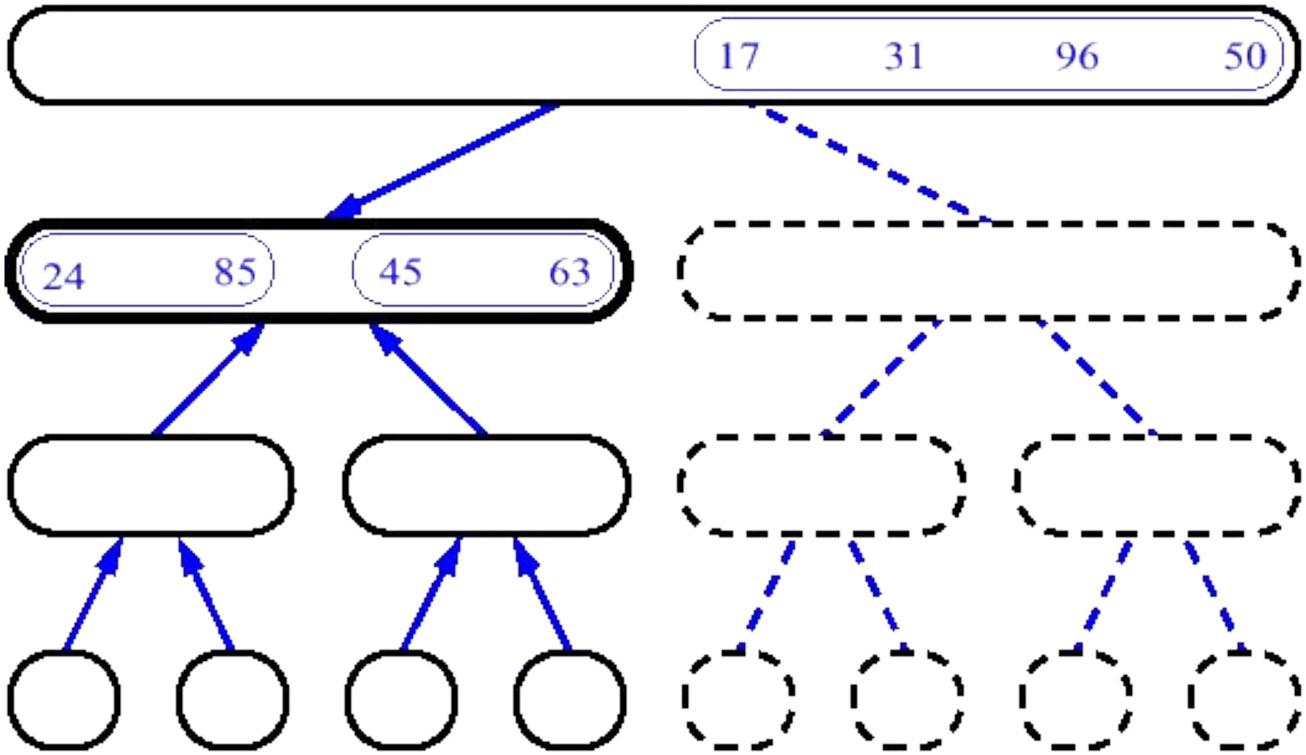
MergeSort (Example) - 14



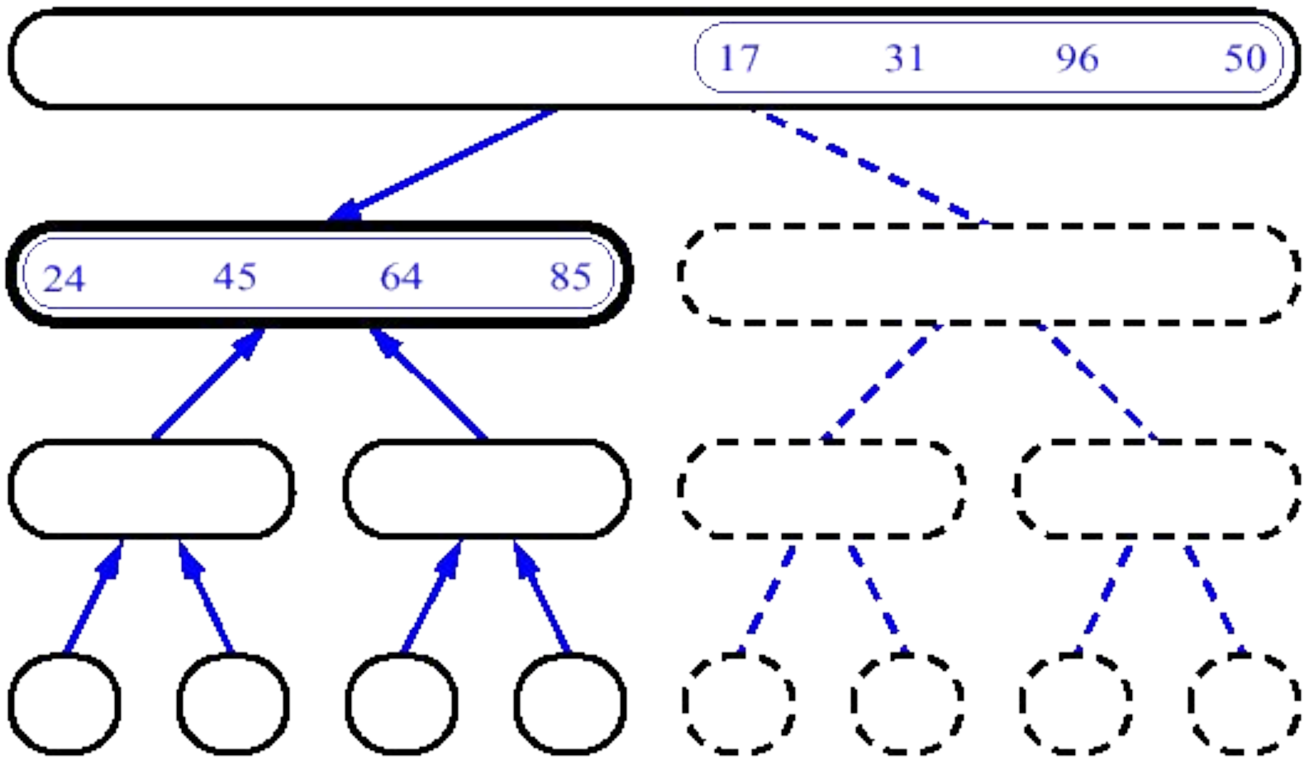
MergeSort (Example) - 15



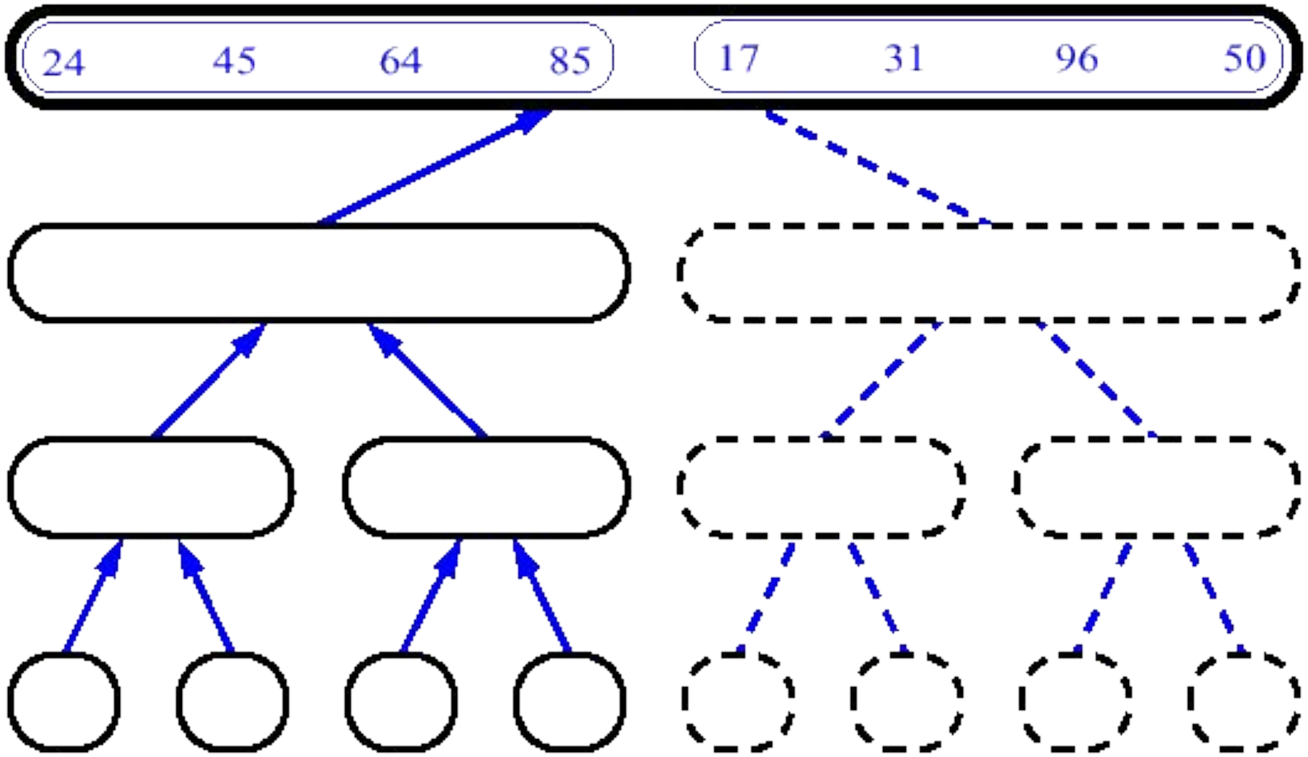
MergeSort (Example) - 16



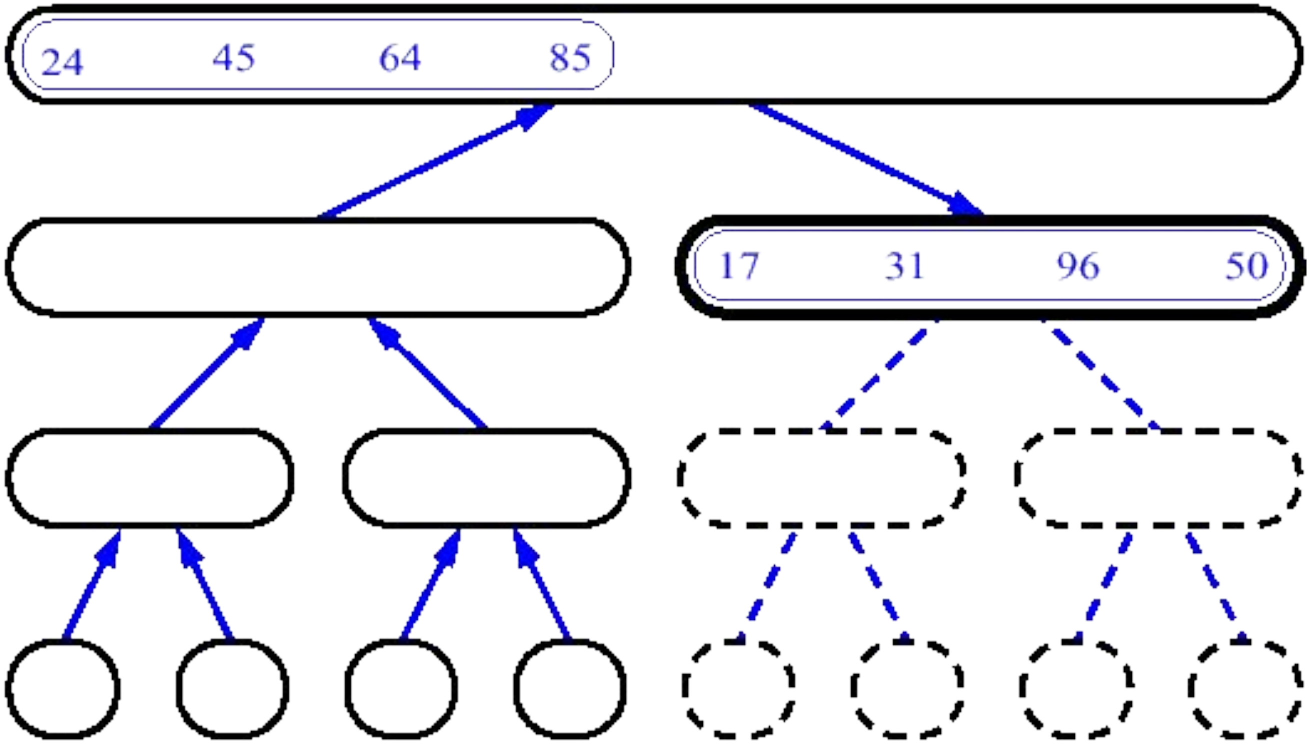
MergeSort (Example) - 17



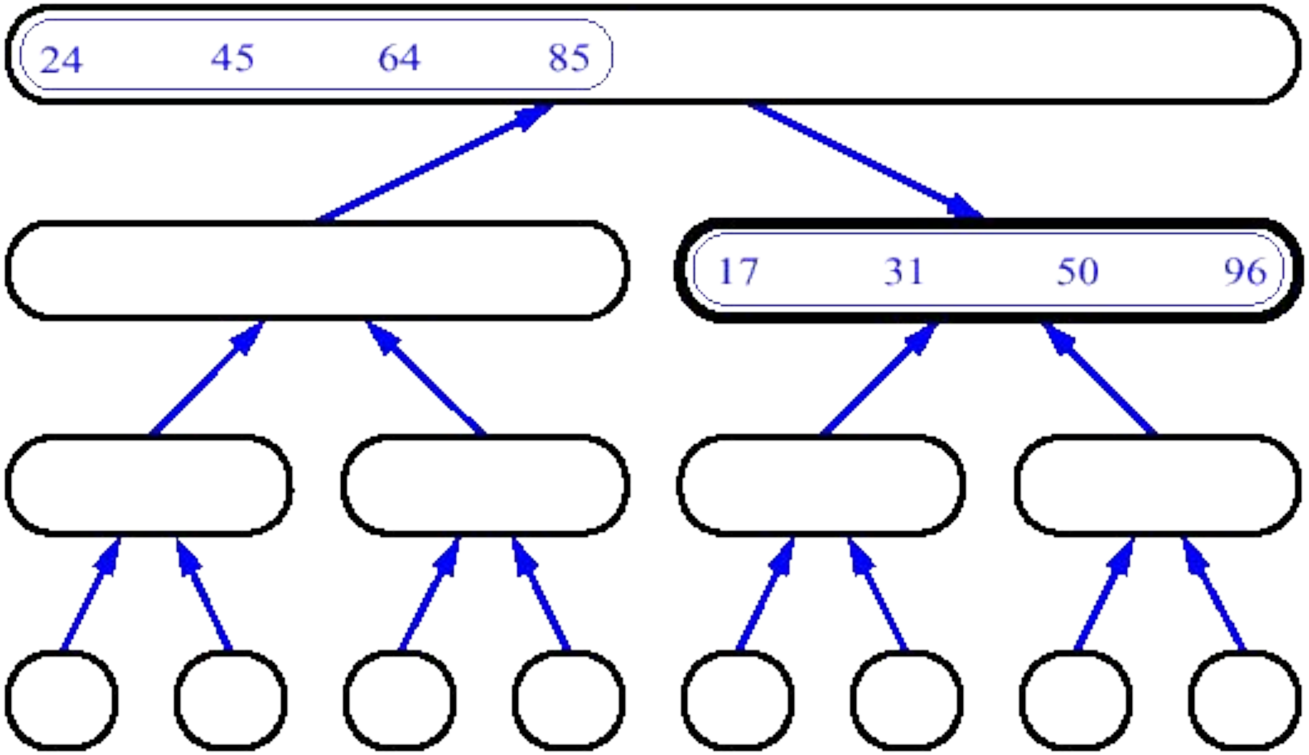
MergeSort (Example) - 18



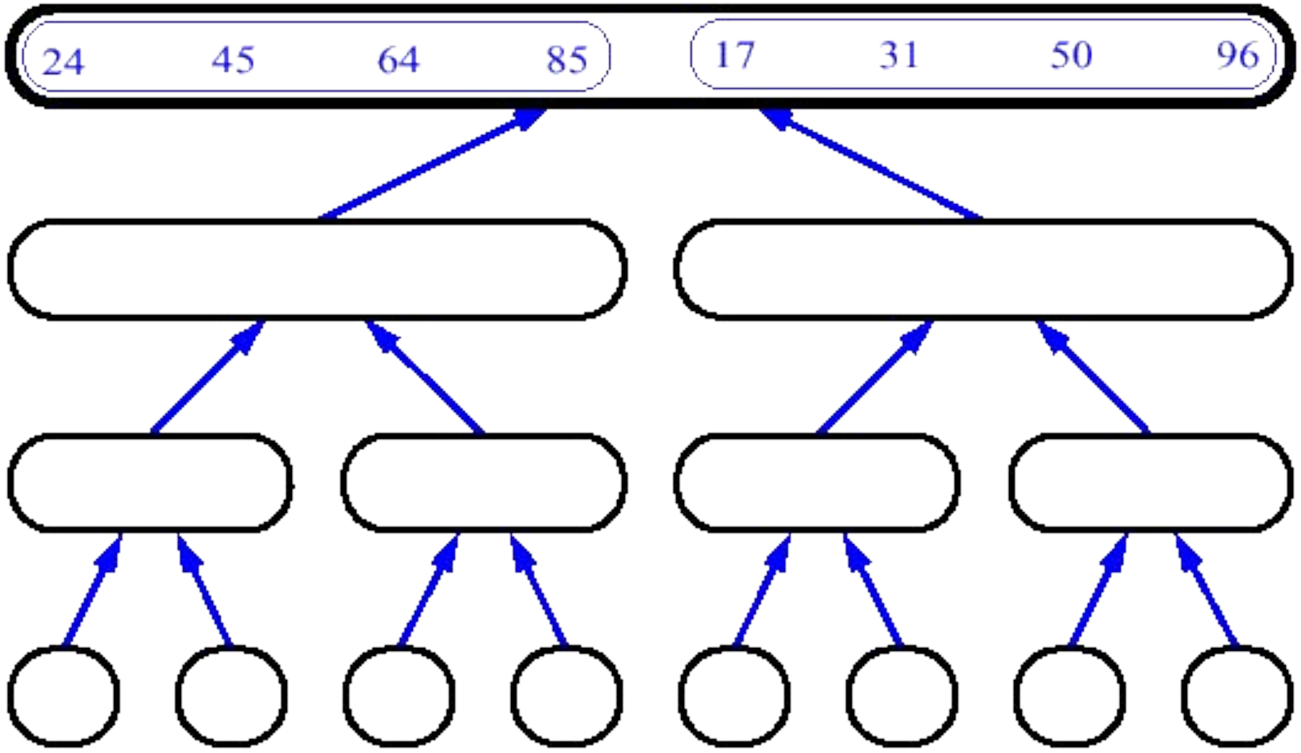
MergeSort (Example) - 19



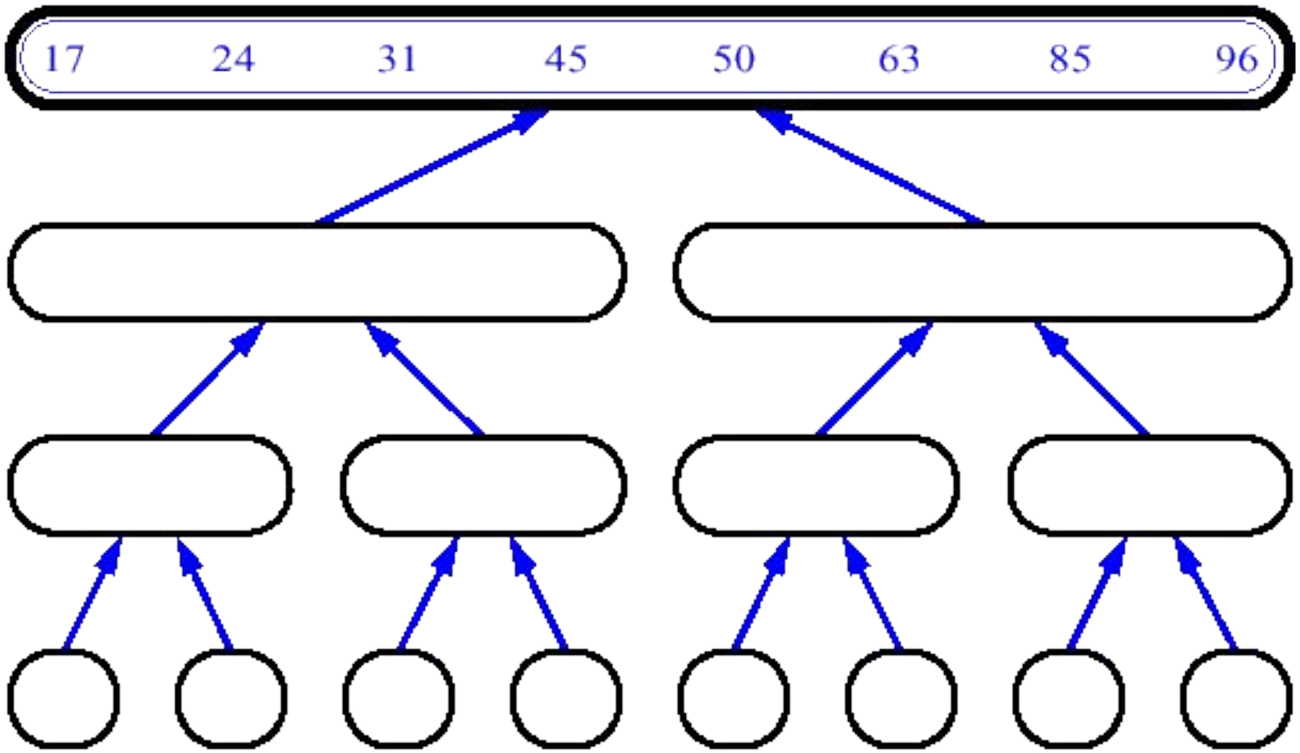
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22

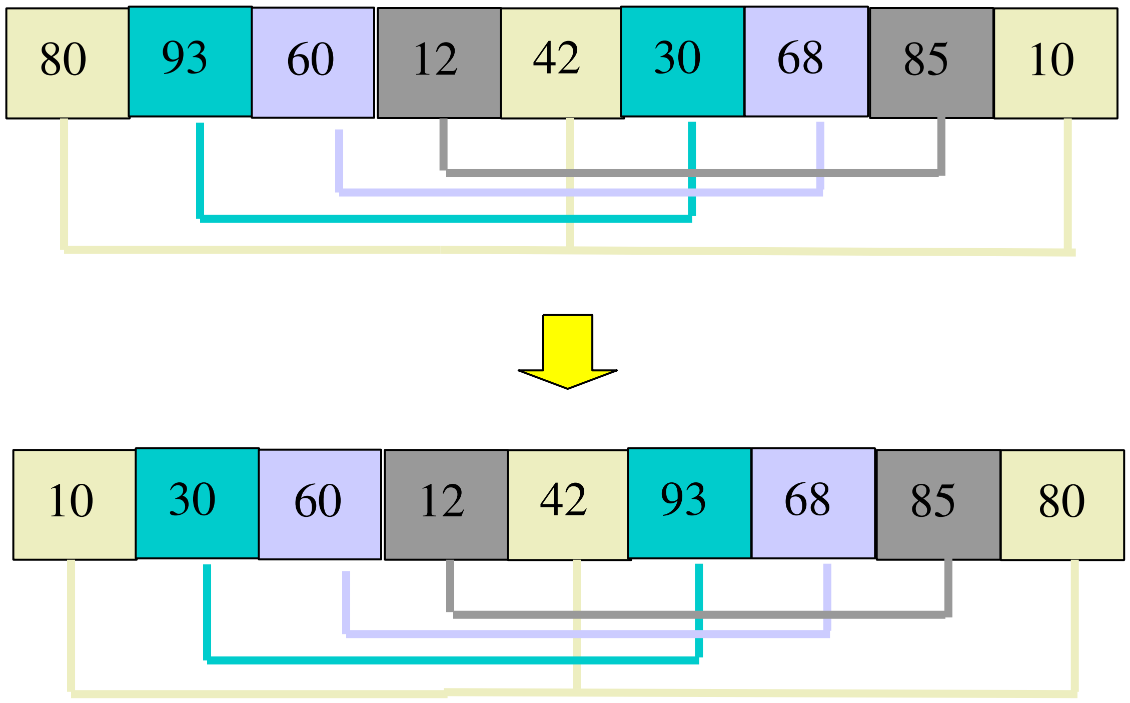


Shell Sort

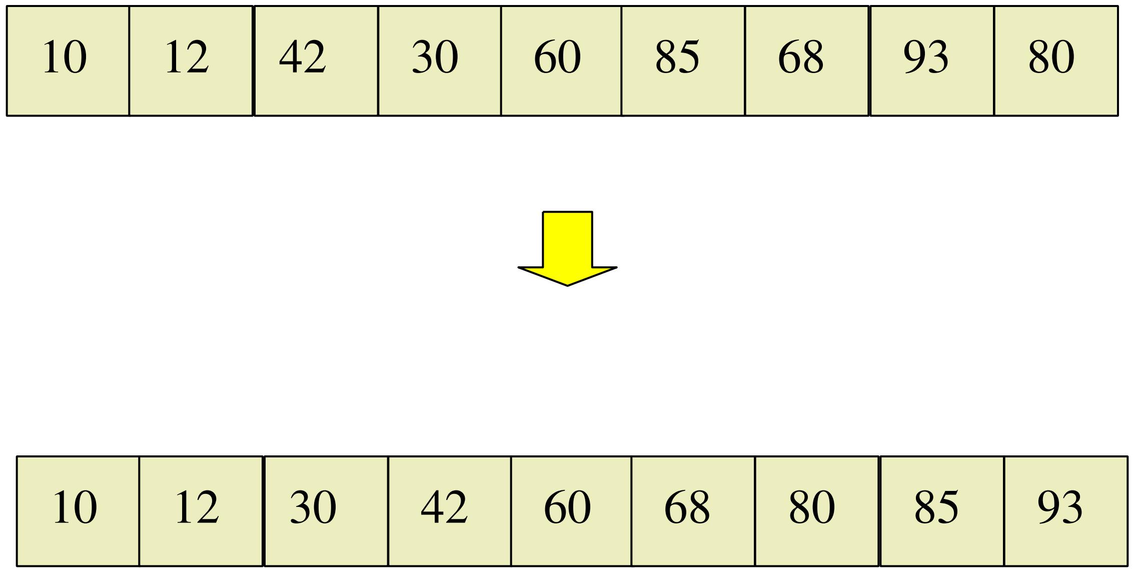
Shell Sort -Background

- General Theory:
  - Makes use of the intrinsic strengths of Insertion sort. Insertion sort is fastest when:
    - The array is nearly sorted.
    - The array contains only a small number of data items.
  - Shell sort works well because:
  - It always deals with a small number of elements.
  - Elements are moved a long way through array with each swap and this leaves it more nearly sorted.

Shell Sort - example



Shell Sort - example (3)



Shellsort Examples

- Sort: 18 32 12 5 38 33 16 2

8 Numbers to be sorted, Shell’s increment will be floor(n/2)

*** floor(8/2) □ floor(4) = 4**

increment 4: **1 2 3 4** (visualize underlining)

18 32 12 5 38 33 16 2

Step **1**) Only look at **18** and **38** and sort in order ;
18 and **38** stays at its current position because they are in order.

Step **2**) Only look at **32** and **33** and sort in order ;
32 and **33** stays at its current position because they are in order.

Step **3**) Only look at **12** and **16** and sort in order ;
12 and **16** stays at its current position because they are in order.

Step **4**) Only look at **5** and **2** and sort in order ;
2 and **5** need to be switched to be in order.

Shellsort Examples (con't)

- Sort: 18 32 12 5 38 33 16 2

Resulting numbers after increment 4 pass:

18 32 12 *2* 38 33 16 *5*

*** floor(4/2) □ floor(2) = 2**

increment 2: 1 2

18 32 12 2 38 33 16 5

Step 1) Look at 18, 12, 38, 16 and sort them in their appropriate location:

12 38 16 2 18 33 38 5

Step 2) Look at 32, 2, 33, 5 and sort them in their appropriate location:

12 2 16 5 18 32 38 33

Shellsort Examples (con't)

- Sort: 18 32 12 5 38 33 16 2

*** floor(2/2) □ floor(1) = 1**

increment 1: 1

12 2 16 5 18 32 38 33

2 5 12 16 18 32 33 38

The last increment or phase of Shellsort is basically an Insertion Sort algorithm.

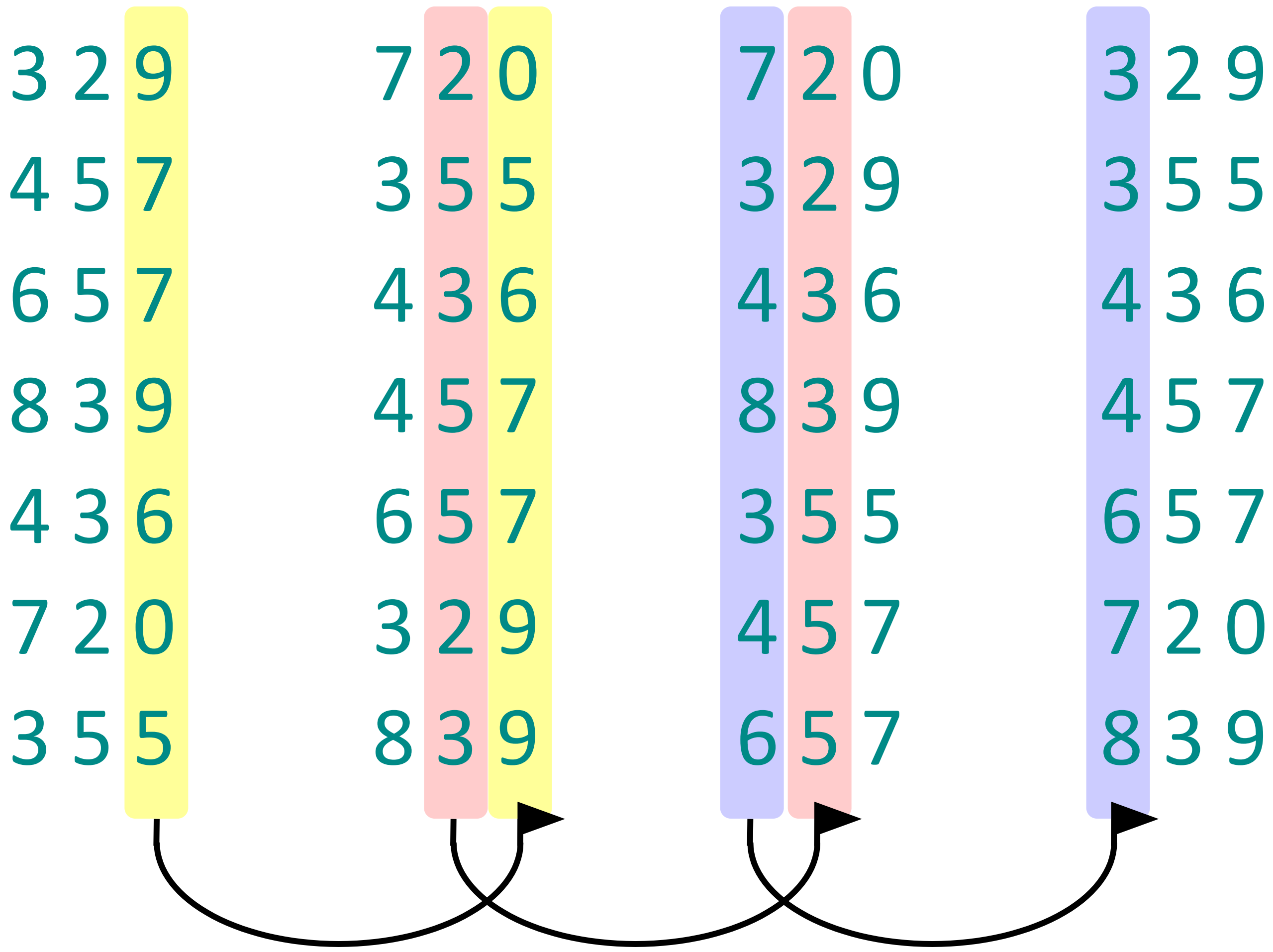
Radix Sort

4.RADIX SORT

- It does not compare the values of the numbers to be sorted.
- Instead it sort by processing the numbers digit by digit, starting from the least significant digit followed by the second least significant digit and so on till the most significant digit.
- Radix sort sorts a set of integers by making several passes over the set , one pass per digit.

- So the maximum number of passes will be equal to the number of passes will be equal to the number of digits in the largest number among the given set of numbers.
- It should be remembered that if during a certain pass,there are several numbers with the same digit value then they will remain at the same relative places.
- Also if there is no significant digit in a number to compare with others then it is assumed to be 0.

Operation of radix sort



COMPLEXITY ANALYSIS

SORT	WORST CASE	AVERAGE CASE	BEST CASE
BUBBLE SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$
SELECTION SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$
INSERTION SORT	$O(n^2)$	$O(n^2)$	$O(n)$
HEAP SORT	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
MERGE SORT	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QUICK SORT	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
RADIX SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$