Sorting Algorithms

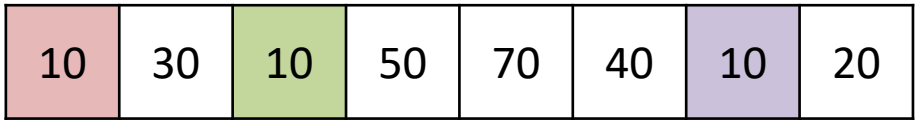
Characteristics

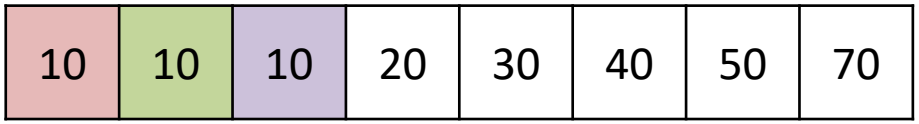
- A sorting algorithm is **In-place** if the algorithm does not use extra space for manipulating the input.
- A sorting algorithm is **stable** if it does not change the order of elements with the same value.
- Online/Offline: The algorithm that accepts a new element while the sorting process is going on, that algorithm is called the online sorting algorithm.

Introduction

- Rearranging elements of an array in some order.
- Various types of sorting are:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Shell Sort
  - Quick Sort
  - Merge Sort
  - Counting Sort
  - Radix Sort
  - Bucket Sort

- In place.
- Stable.
- Online.



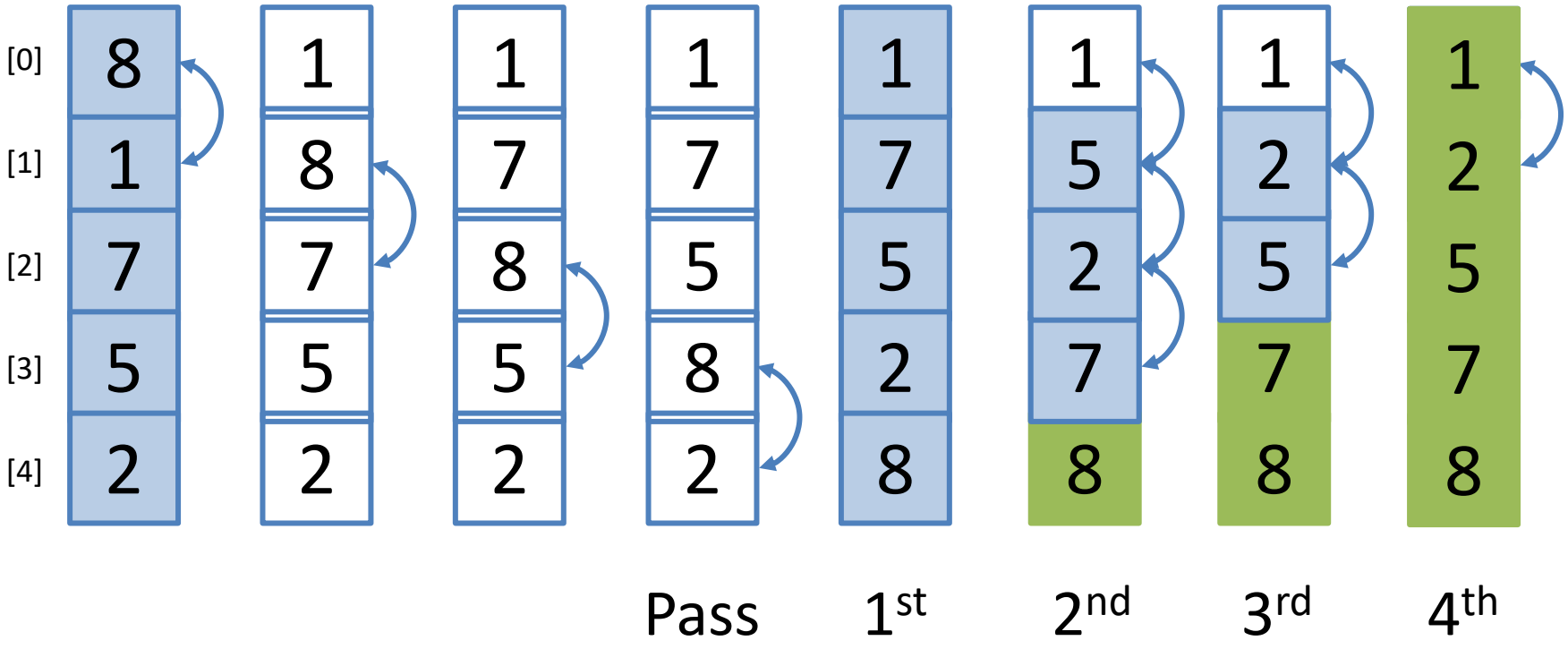


Sorting Algorithms	In - Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Quick Sort	Yes	No
Merge Sort	No (because it requires an extra array to merge the sorted subarrays)	Yes
Heap Sort	Yes	No

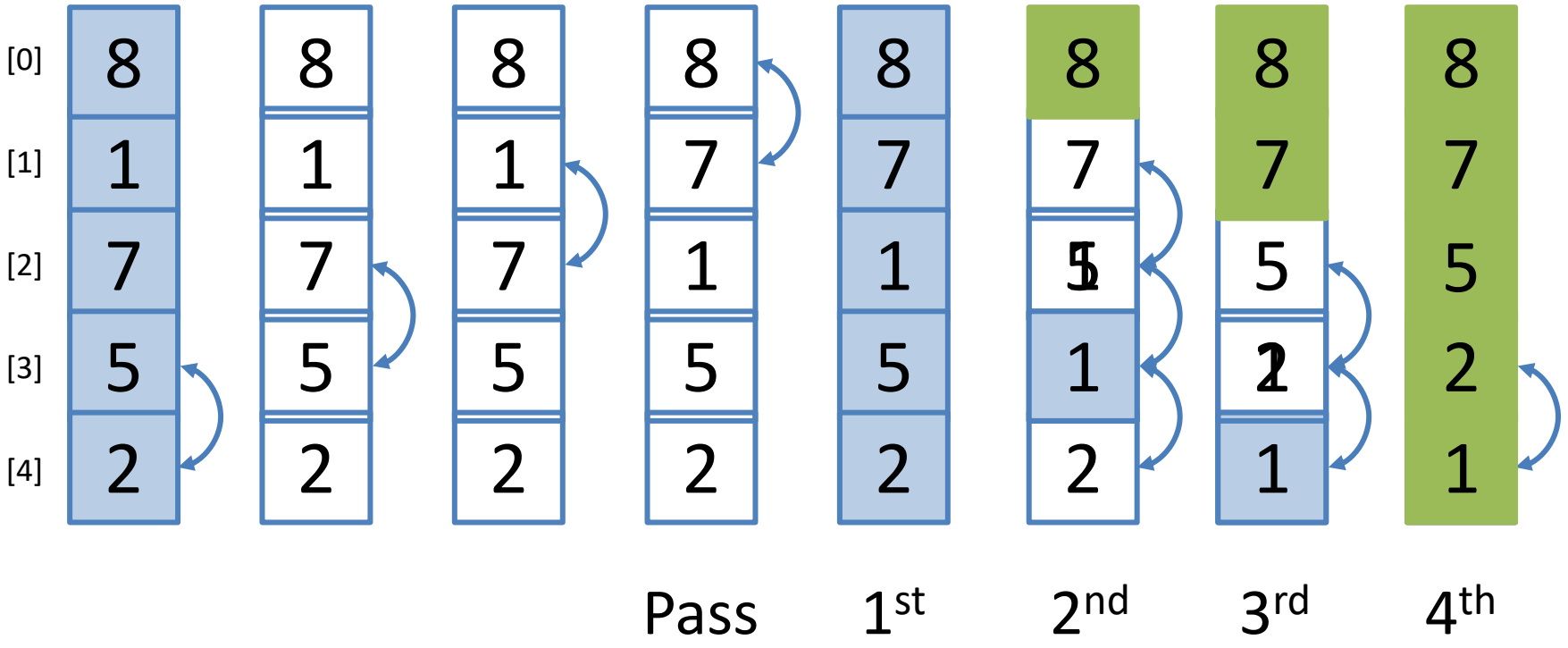
From, the above sorting algorithms, the insertion sort is online

Bubble Sort

Bubble Sort – Ascending



Bubble Sort – Descending



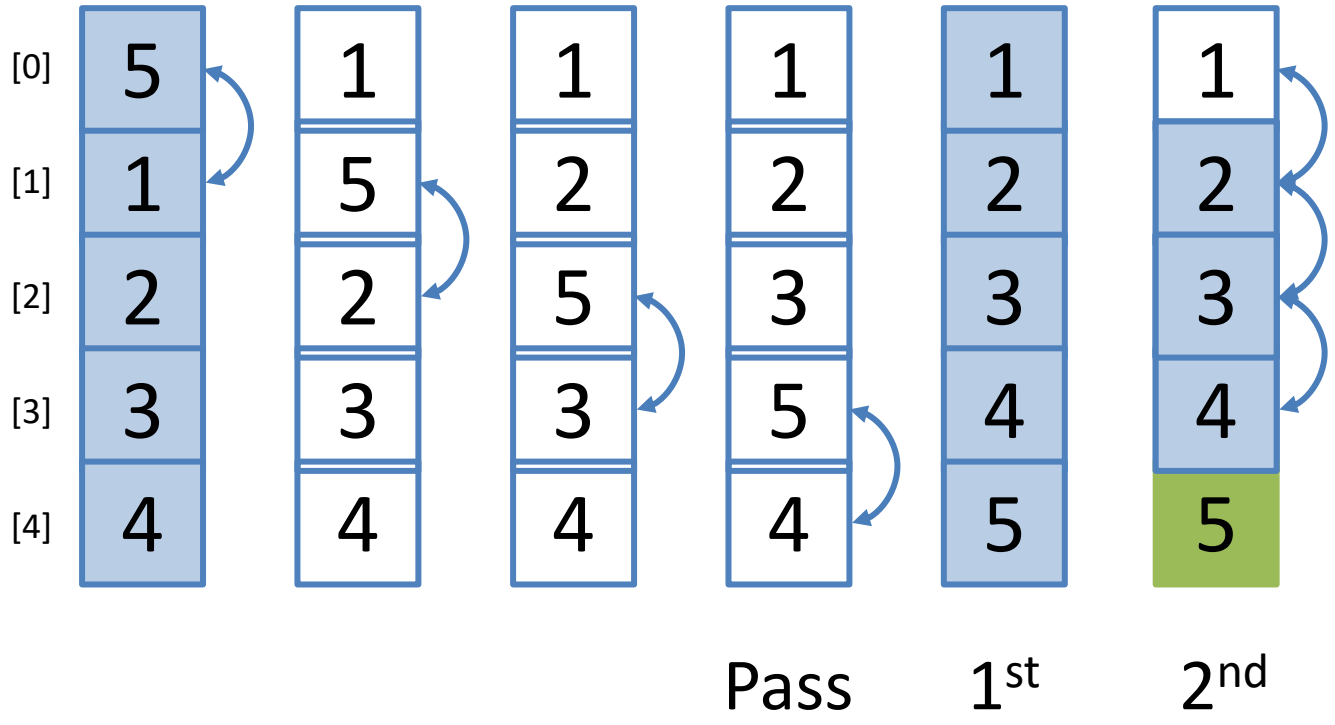
Algorithm – Bubble Sort

Algorithm bubbleSort(A,n)
Input: An array **A** containing **n** integers.
Output: The elements of **A** get sorted in increasing order.

1. **for** i = 1 to n – 1 **do**
2. **for** j = 0 to n - i – 1 **do**
3. **if** A[j] > A[j + 1]
4. Exchange A[j] with A[j+1]

In all the cases, complexity is of the order of n^2.

Optimized Bubble Sort?



Algorithm – Optimized Bubble Sort

Algorithm bubbleSortOpt(A,n)
Input: An array **A** containing **n** integers.
Output: The elements of **A** get sorted in increasing order.

```
1.  for  i = 1  to  n – 1
2.      flag = true
3.      for j = 0  to  n – i – 1  do
4.          if A[j] > A[j + 1]
5.              flag = false
6.              Exchange A[j] with A[j+1]
7.      if  flag == true
8.          break;
```

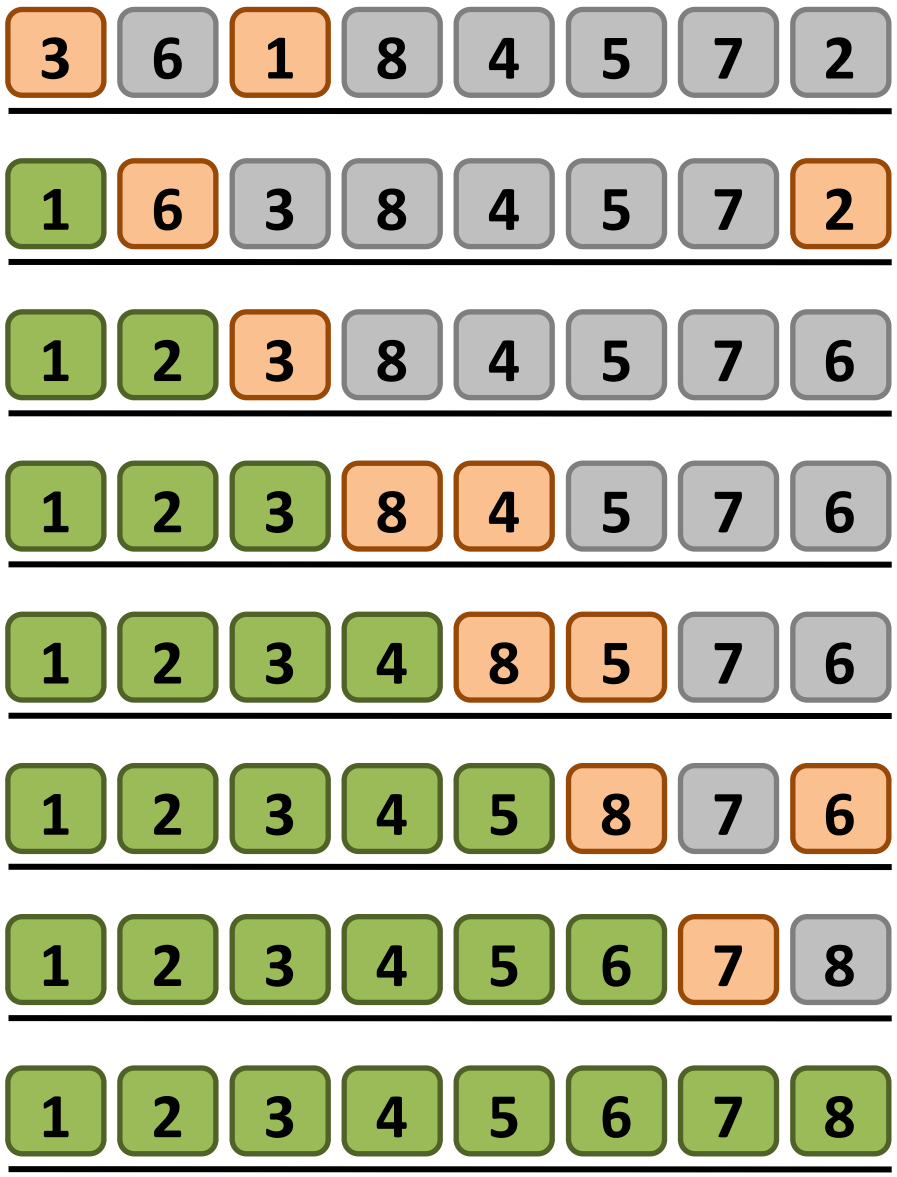
The best case complexity reduces to the order of n, but the worst and average is still n^2. So, overall the complexity is of the order of n^2 again.

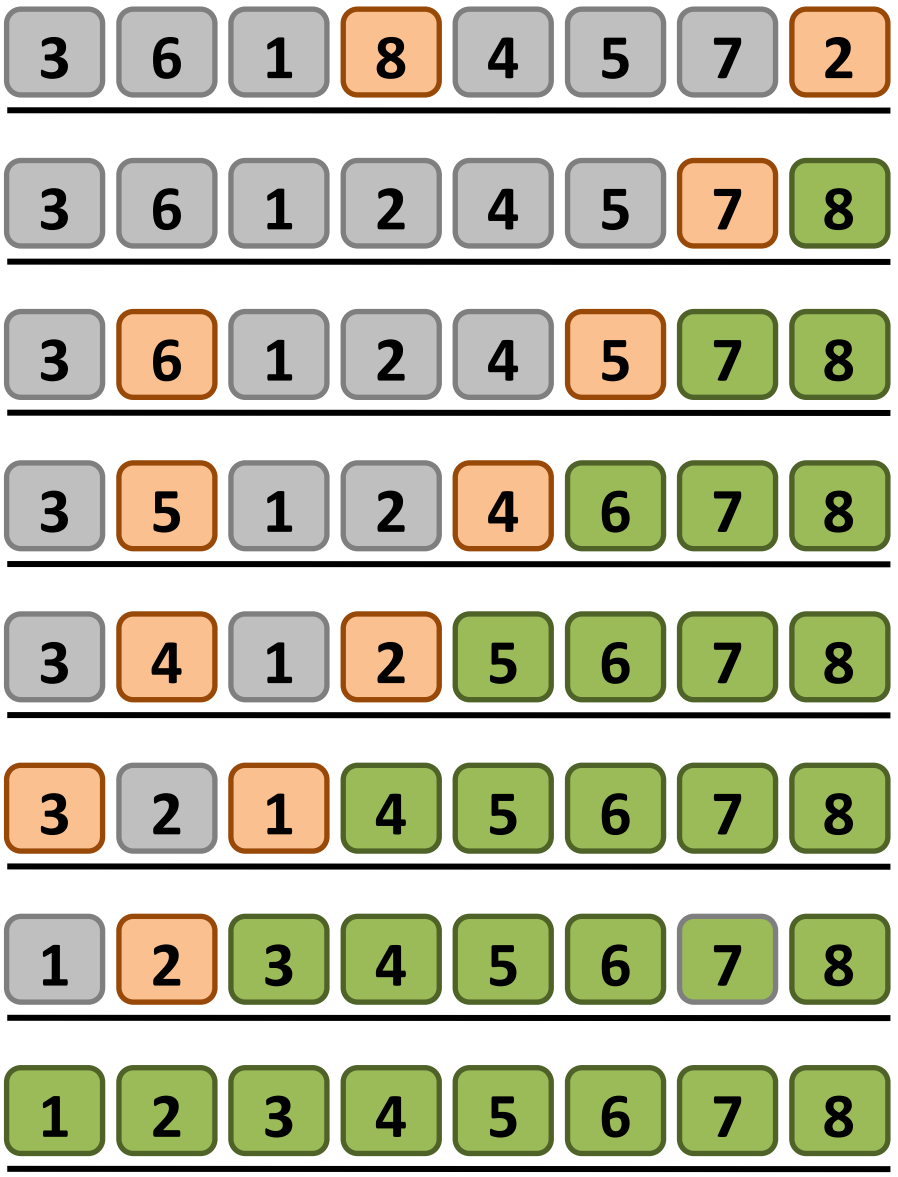
Selection Sort

Selection Sort

- In-place comparison-based algorithm.
- Divides the list into two parts
  - The sorted part, which is built up from left to right at the front (left) of the list, and
  - The unsorted part, that occupy the rest of the list at the right end.
- The algorithm proceeds by
  - Finding the smallest (or the largest) element in the unsorted array
  - Swapping it with the leftmost (or the rightmost) unsorted element
  - Moving the boundary one element to the right.
  - This process continues till the array gets sorted.
- Not suitable for large data sets.
- Complexity is $O(n^2)$, where n is the number of elements.

Example


3 6 1 8 4 5 7 2
1 6 3 8 4 5 7 2
1 2 3 8 4 5 7 6
1 2 3 8 4 5 7 6
1 2 3 4 8 5 7 6
1 2 3 4 5 8 7 6
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8


3 6 1 8 4 5 7 2
3 6 1 2 4 5 7 8
3 6 1 2 4 5 7 8
3 5 1 2 4 6 7 8
3 4 1 2 5 6 7 8
3 2 1 4 5 6 7 8
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

Algorithm

- **Algorithm selectionSort(a[], n)**
- Input: An array **a** containing **n** elements.
- Output: The elements of **a** get sorted in increasing order.

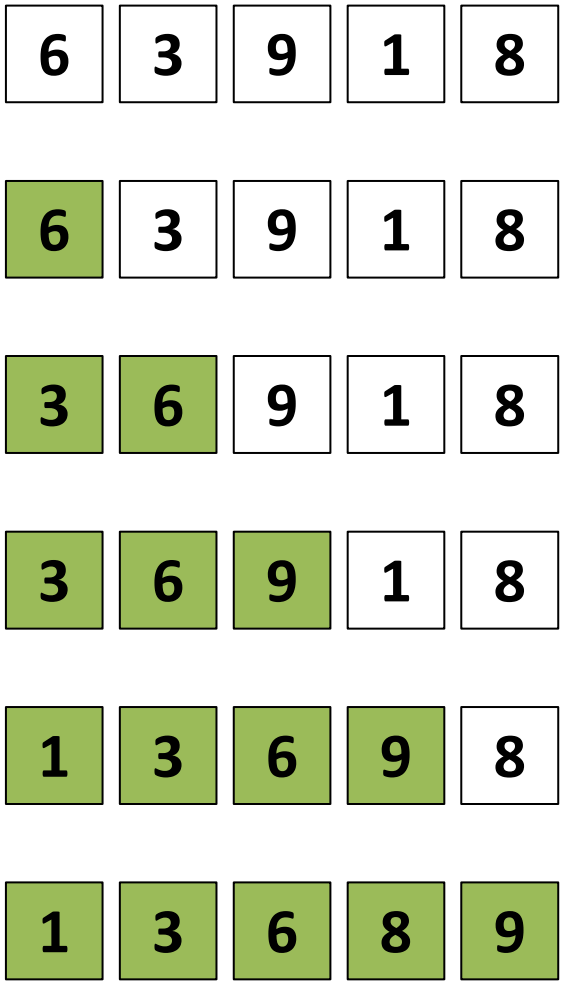
```
1.  for i = 0 to n – 2
2.          min = i
3.          for j = i+1 to n – 1
4.                  if a[j] < a[min]
5.                          min = j
6.          if min != i
7.                  Exchange a[min] with a[i]
```

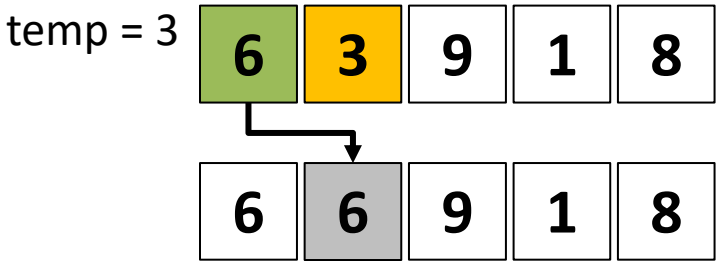
Insertion Sort

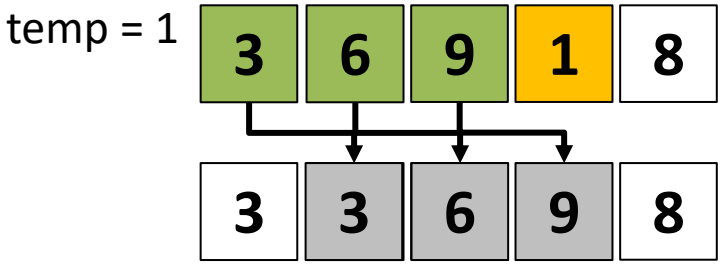
Insertion Sort

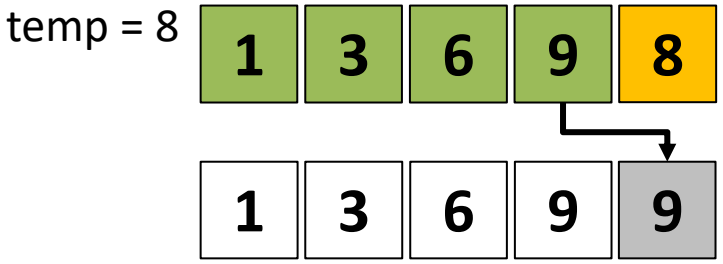
- An in-place, stable, online comparison-based sorting algorithm.
- Always keeps the lower part of an array in the sorted order.
- A new element will be inserted in the sorted part at an appropriate place.
- The algorithm searches sequentially, move the elements, and inserts the new element in the array.
- Not suitable for large data sets
- Complexity is $\boldsymbol{O(n^2)}$, where n is the number of elements.
- Best case complexity is $\boldsymbol{O(n)}$.

Example


6 3 9 1 8
6 3 9 1 8
3 6 9 1 8
3 6 9 1 8
1 3 6 9 8
1 3 6 8 9


temp = 3
6 3 9 1 8
6 6 9 1 8


temp = 1
3 6 9 1 8
3 3 6 9 8


temp = 8
1 3 6 9 8
1 3 6 9 9

Algorithm

- **Algorithm insertionSort(a[], n)**
- Input: An array **a** containing **n** elements.
- Output: The elements of **a** get sorted in increasing order.

```
1.  for i = 1 to n – 1
2.          temp = a[i]
3.          j = i
4.          while j > 0 and a[j-1] > temp
5.                  a[j] = a[j-1]
6.                  j = j – 1
7.          a[j] = temp
```

Shell Sort

Shell Sort

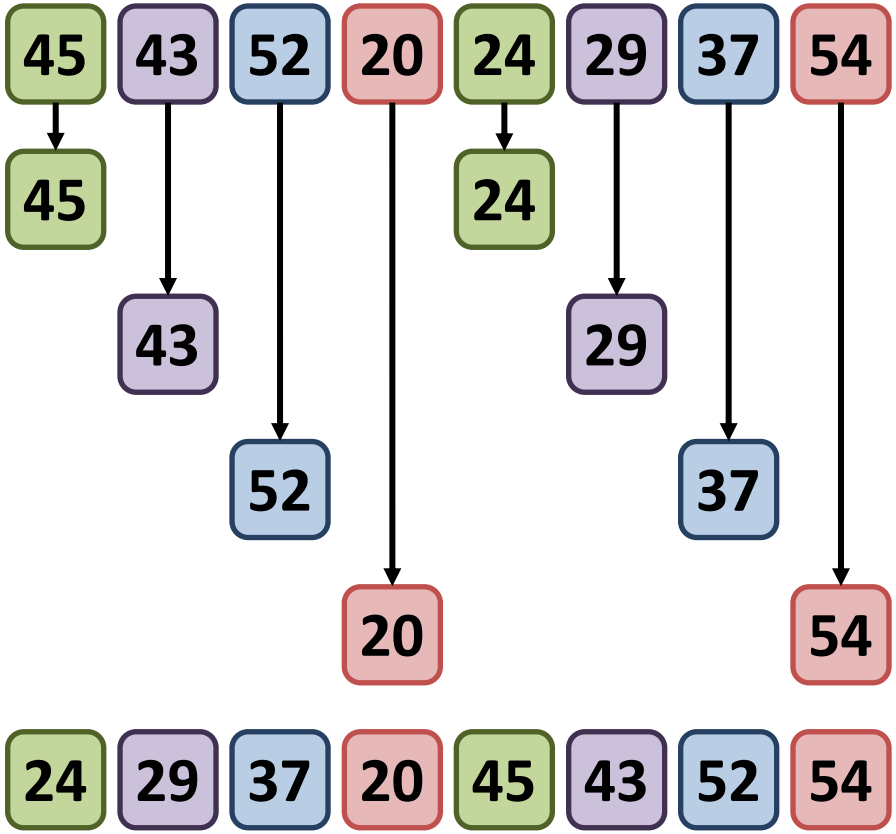
- Improved or generalized insertion sort.
- An in–place comparison sort.
- Also known as diminishing increment sort.
- Breaks the original list into a number of smaller sublists, each of which is sorted using an insertion sort.
- Instead of breaking the list into sublists of contiguous items, the algorithm uses a unique way to choose the sublists.
  - An increment (say h), sometimes called the gap or the interval.
  - A sublist contains all the elements that are i elements apart.
- Complexity lies in between O(n) and $O(n^2)$. Still an open problem.

Contd…

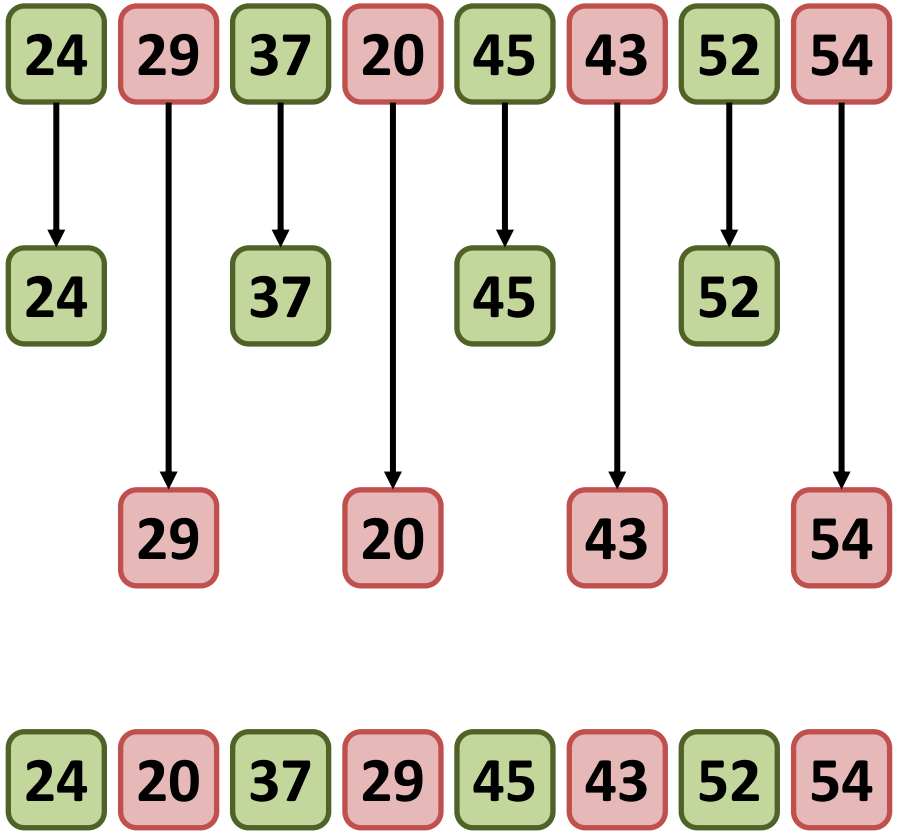
- Increment Sequences:
  - Shell's original sequence: N/2 , N/4 , …, 1
  - Hibbard's increments: 1, 3, 7, …, $2^k - 1$
  - Knuth's increments: 1, 4, 13, …, (3k + 1)
  - Sedgewick's increments: 1, 5, 19, 41, 109, ….
    - Merging of $(9 \times 4^i) - (9 \times 2^i) + 1$ and $4^i - (3 \times 2^i) - 1$
- Start with higher intervals and then reduce the interval after each pass as per the chosen sequence.

Example

- Using original sequence
  - h = N/2 = 8/2 = 4.



  - h = h/2 = 4/2 = 2.



Contd…

– h =14 h/2 = 2/2 = 1.

Algorithm

- **Algorithm shellSort(a[], n)**
- Input: An array **a** containing **n** elements.
- Output: The elements of **a** get sorted in increasing order.

```
1. for (gap = n/2; gap > 0; gap /=2)
2. { for (i = gap; i < n; i++)
3.     { temp = a[i]
4.     for (j = i; j >= gap && a[j - gap] > temp; j -= gap)
5.             a[j] = a[j-gap]
6.     a[j]= temp;          }
7. }
```

Thank You