CSCI 132:
Basic Data Structures and Algorithms

Sorting (Part 4)

Reese Pearsall

Spring 2025

Announcements

- Program 5 due May 4th
- Rubber Duck Extra Credit posted
- Wednesday 4/30 is an optional help session (no lecture)

real

- Program 4 and 5 grading
- Program Testing

Running Time of Sorting Algorithms

	Brief Description	Running Time
Bubble Sort	???	???
Selection Sort	???	???
Merge Sort	???	???
Quick Sort	???	???

```
public int[] selectionSort(int[] array) {
        int n = array.length;
        for(int i = 0; i < n -1; i++) {
                int min_index_so_far = i;
                for (int j = i + 1; j < n; j++) {
                        if(array[j] < array[min_index_so_far]) {
                                min_index_so_far = j;
                        }
                }
                int temp = array[i];
                array[i] = array[min_index_so_far];
                array[min_index_so_far] = temp;
        }
        return array;
}
```

Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

38	27	43	3	9	82	10	14

Insertion Sort

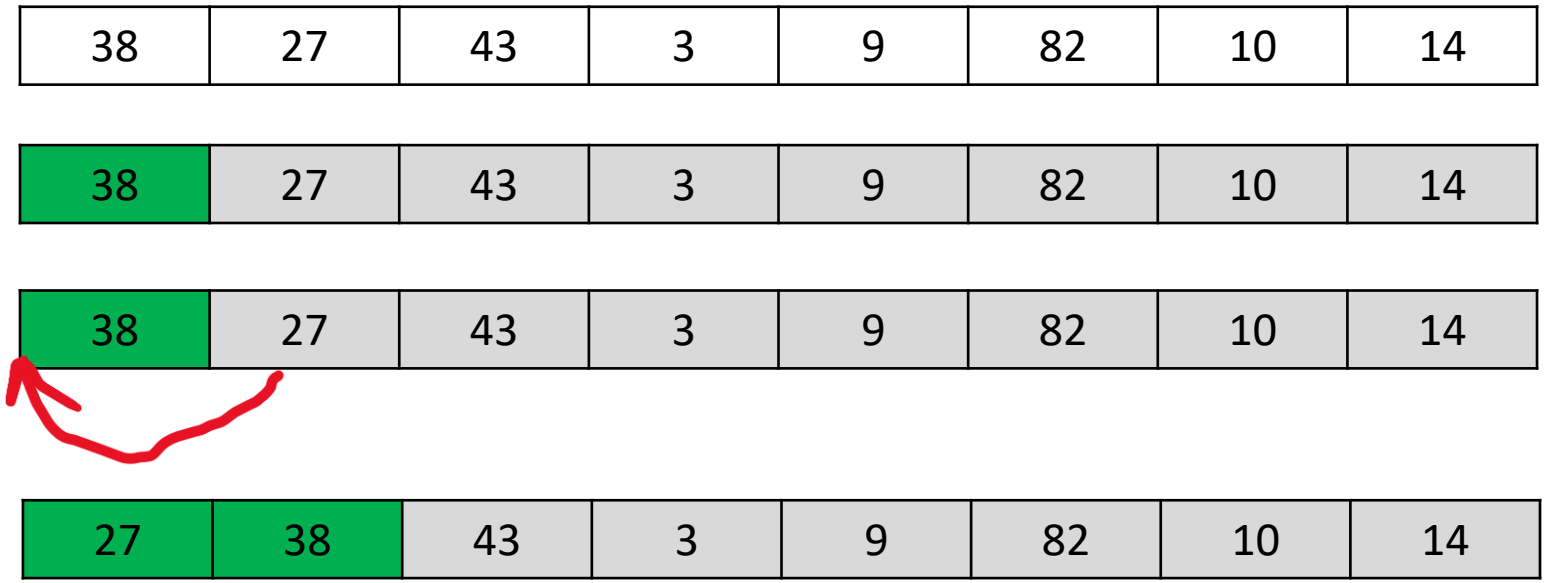
We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

38	27	43	3	9	82	10	14

38	27	43	3	9	82	10	14

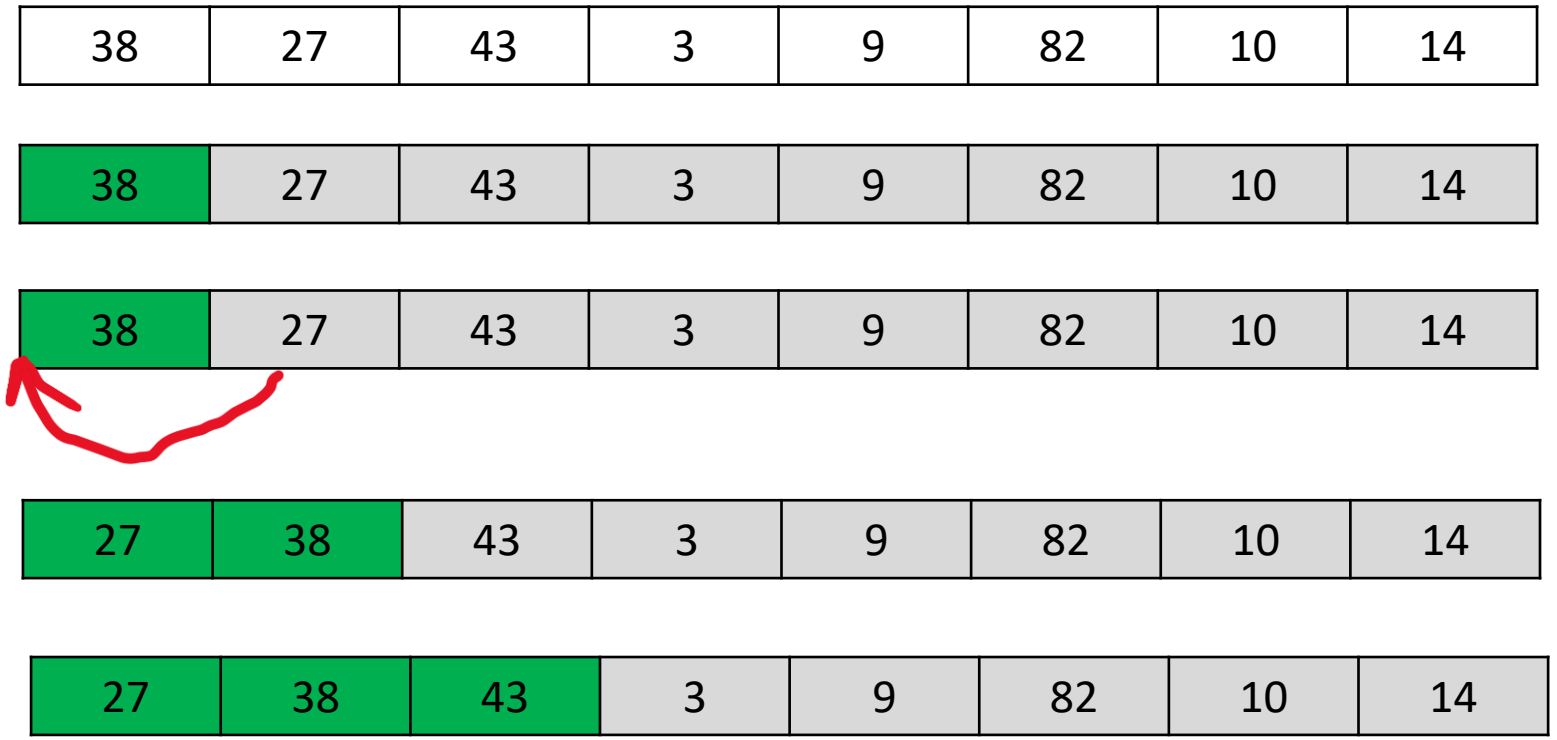
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



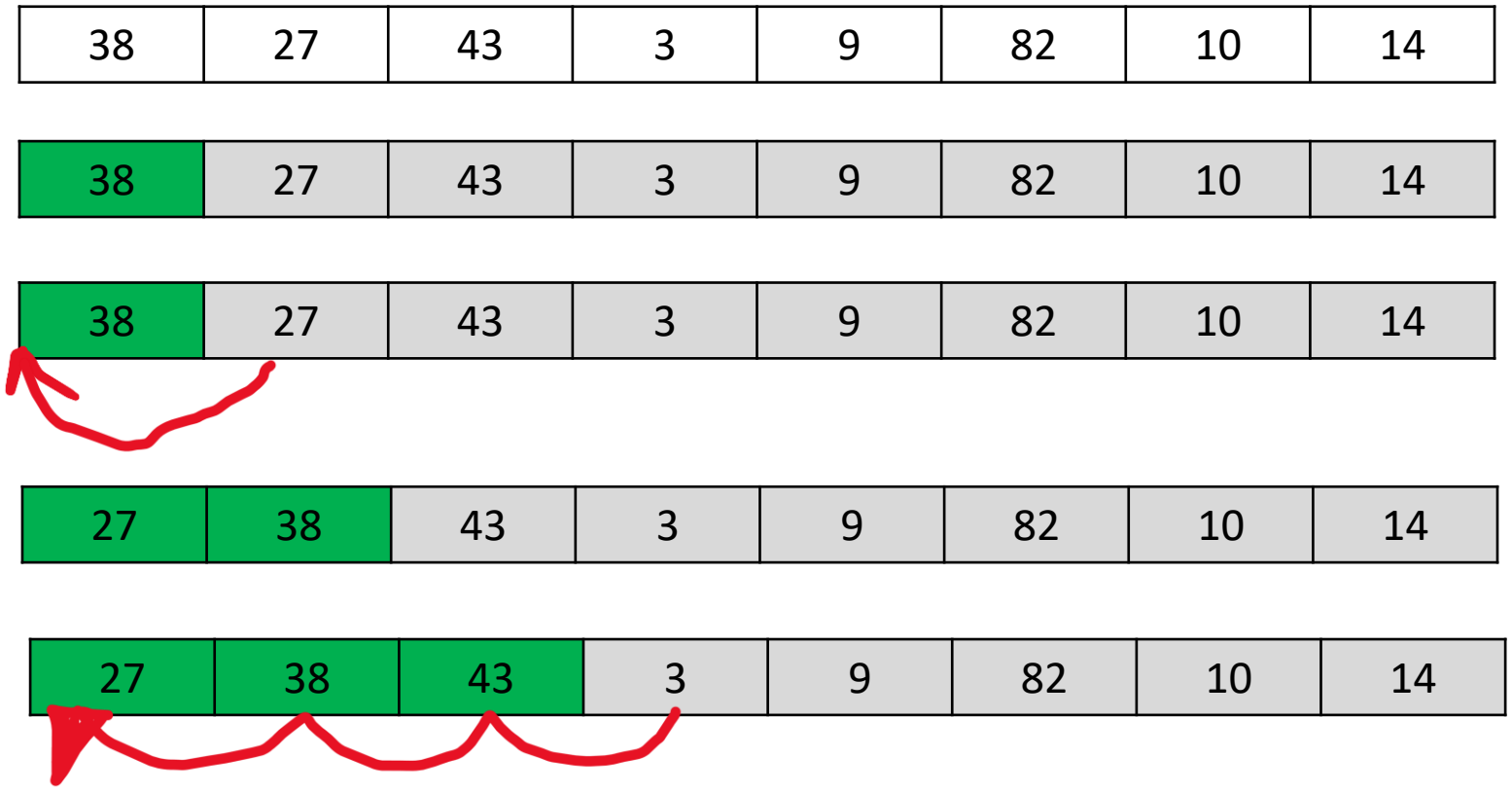
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



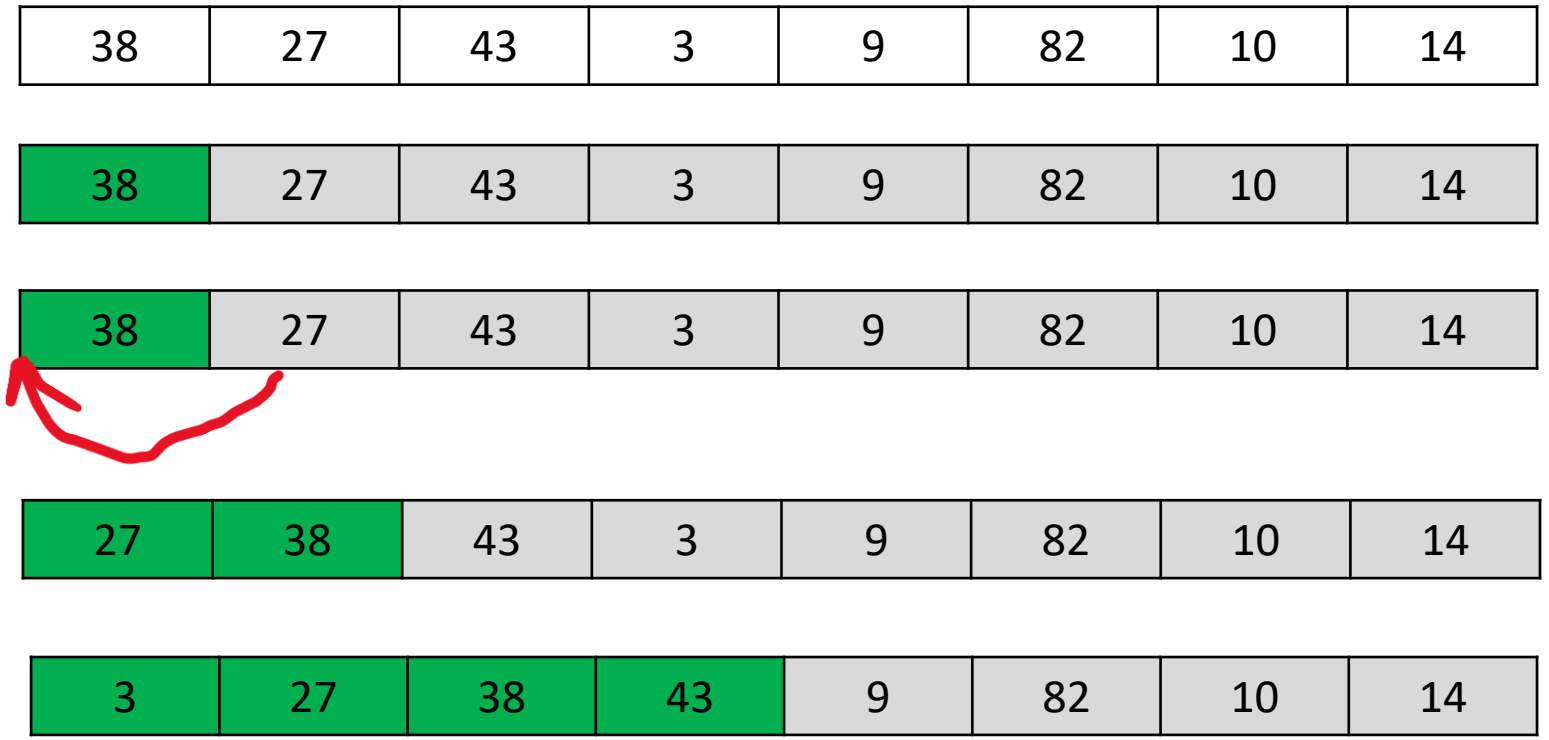
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



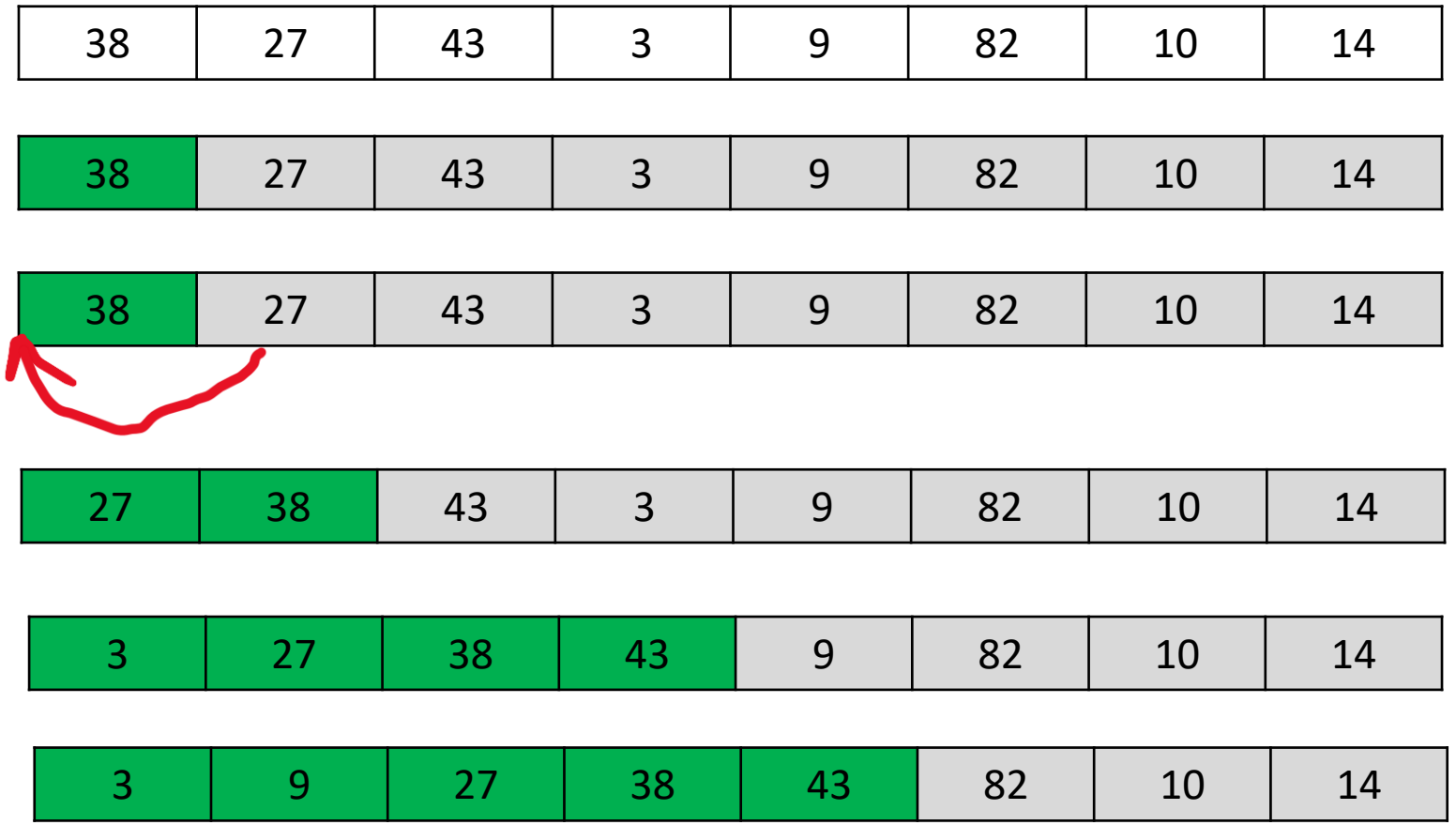
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



Insertion Sort

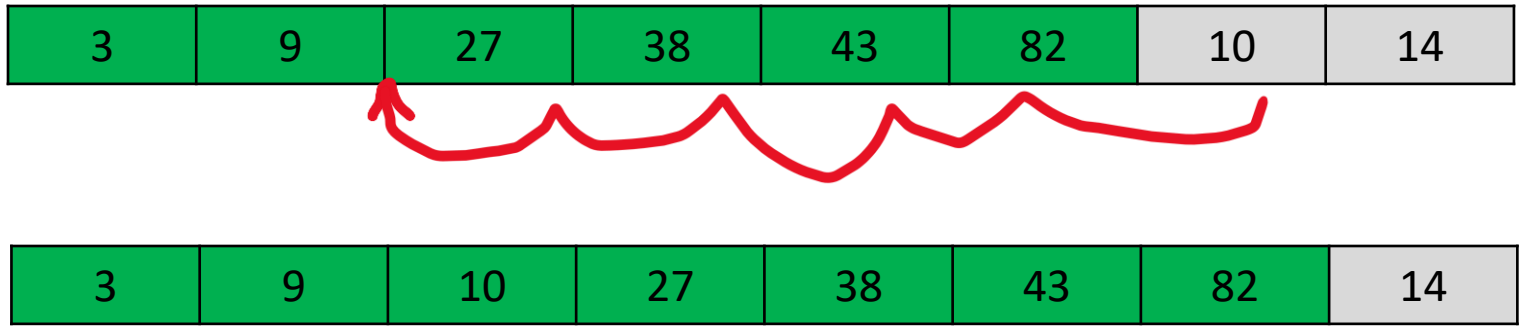
We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

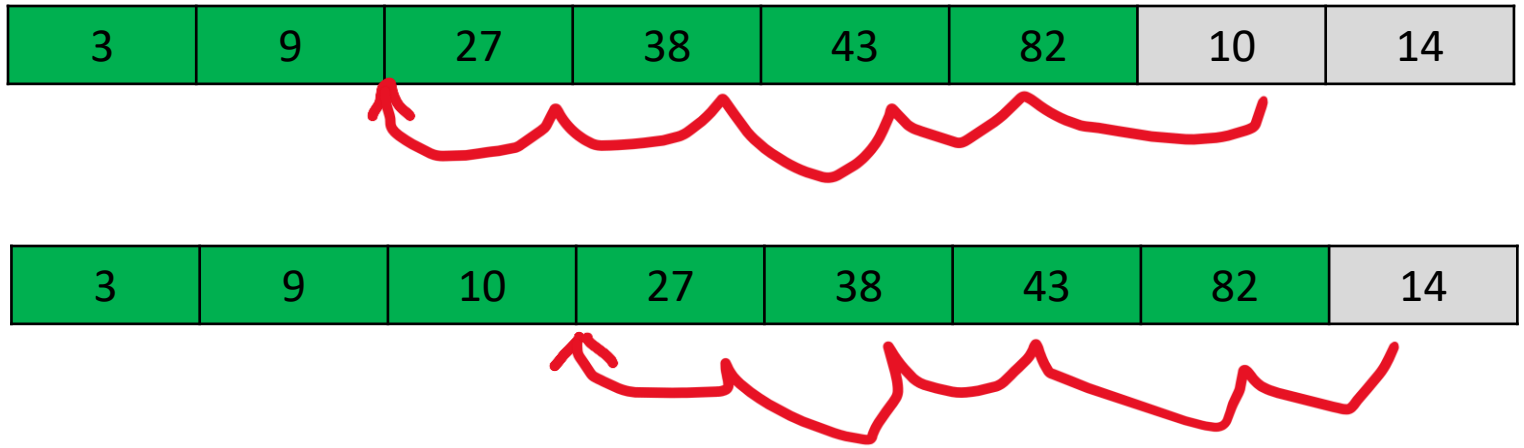
Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



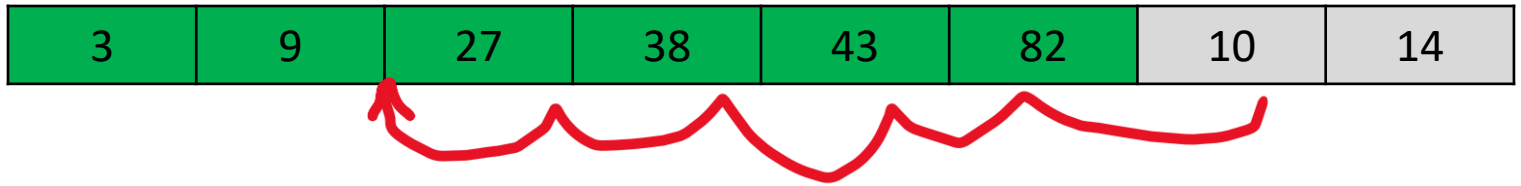
Insertion Sort

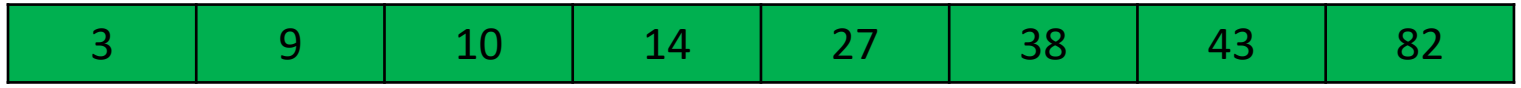
We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section



Insertion Sort

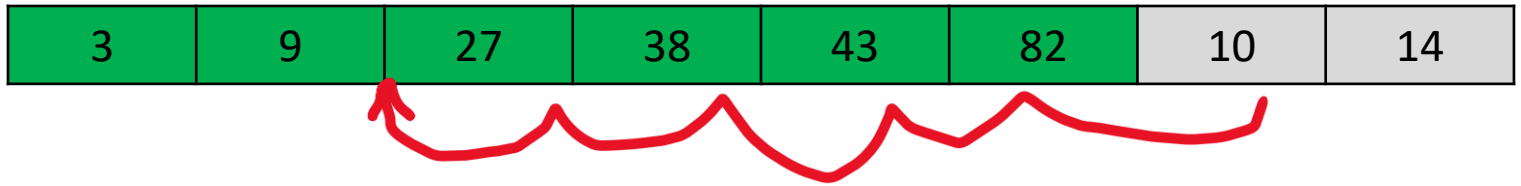
We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

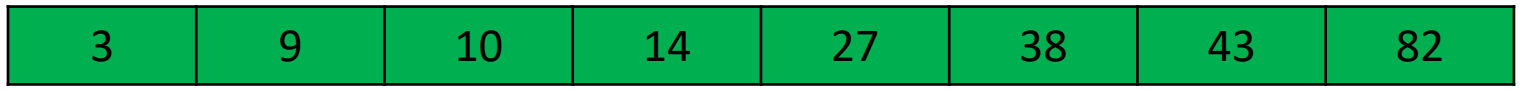




Insertion Sort

We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section





Running time: $O(n^2)$

Insertion Sort

```
void insertionSort(int array[]) {
        int size = array.length;
        for (int step = 1; step < size; step++) {
                int key = array[step];
                int j = step - 1;
                // Compare key with each element on the left of it until an element smaller than
                // it is found.
                // For descending order, change key<array[j] to key>array[j].
                while (j >= 0 && key < array[j]) {
                        array[j + 1] = array[j];
                        --j;
                }
                // Place key at after the element just smaller than it.
                array[j + 1] = key;
        }
}
```

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

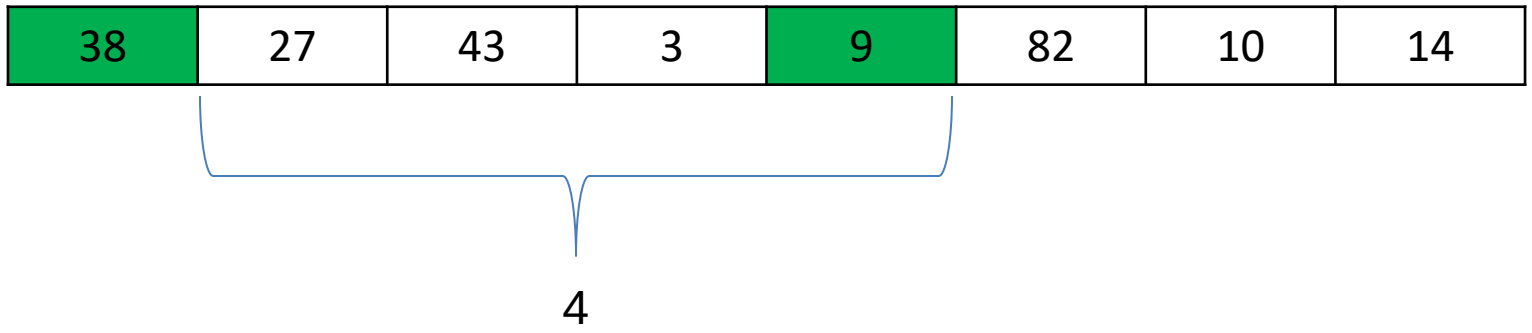
38	27	43	3	9	82	10	14

N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

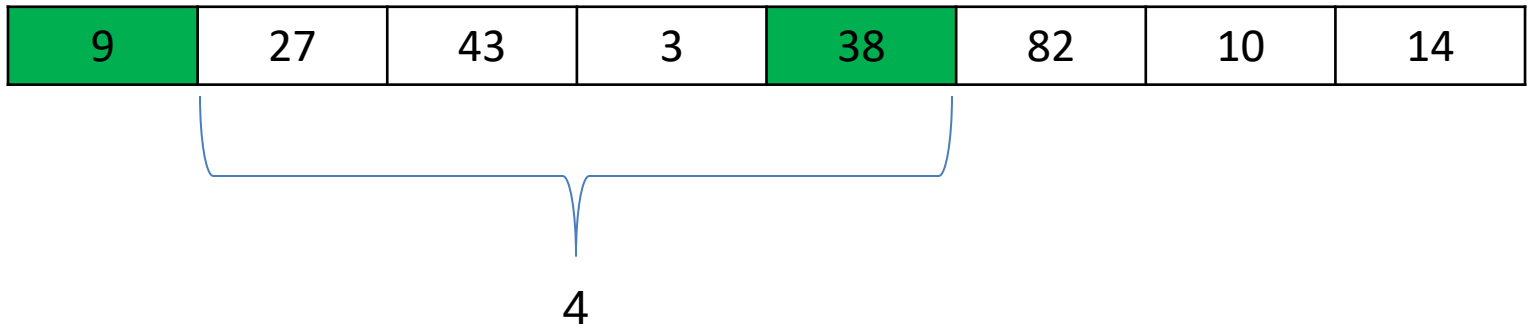


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

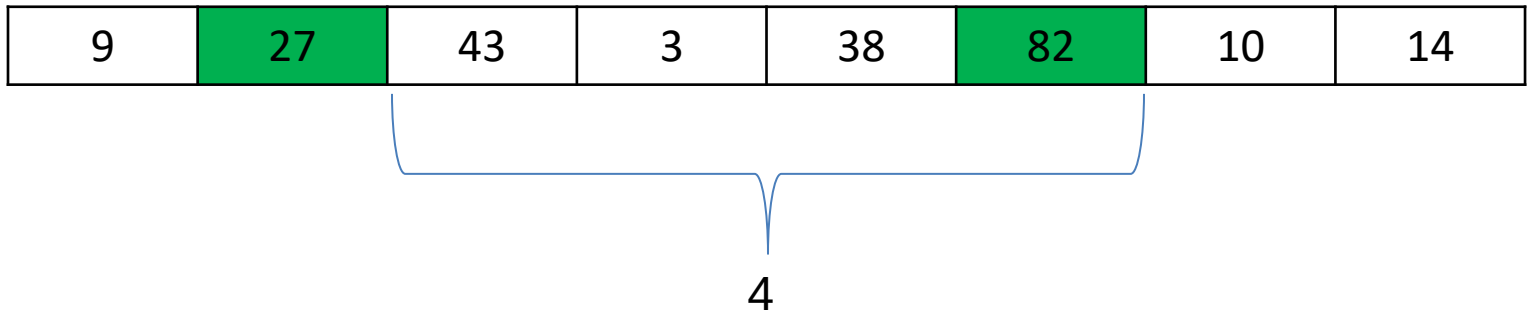


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

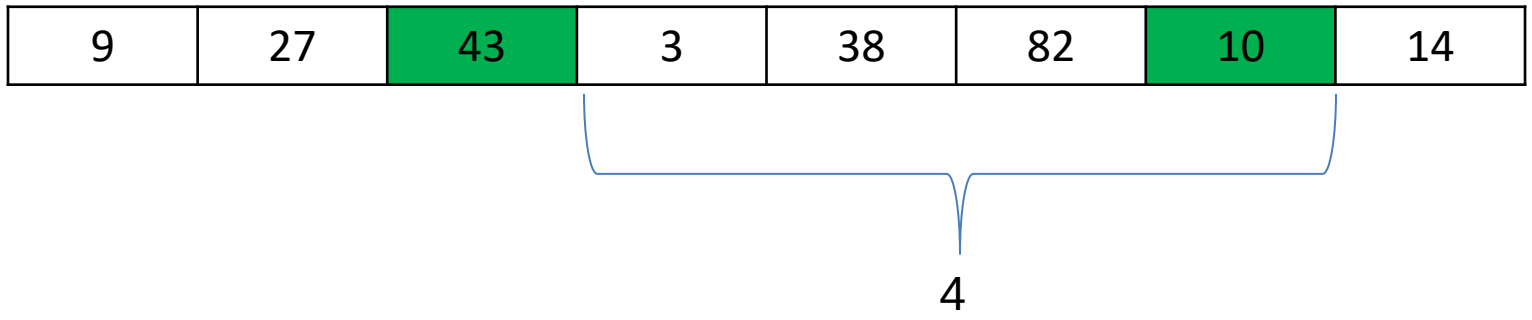


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

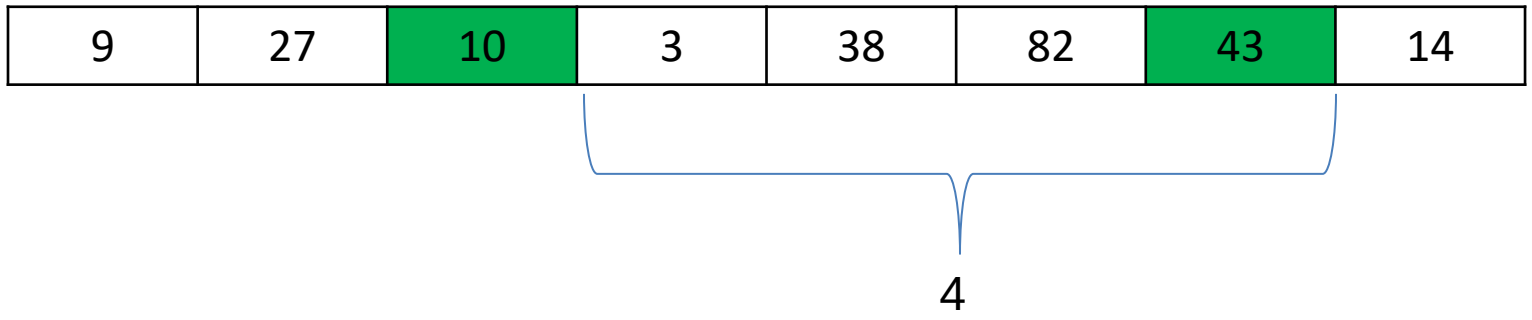


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

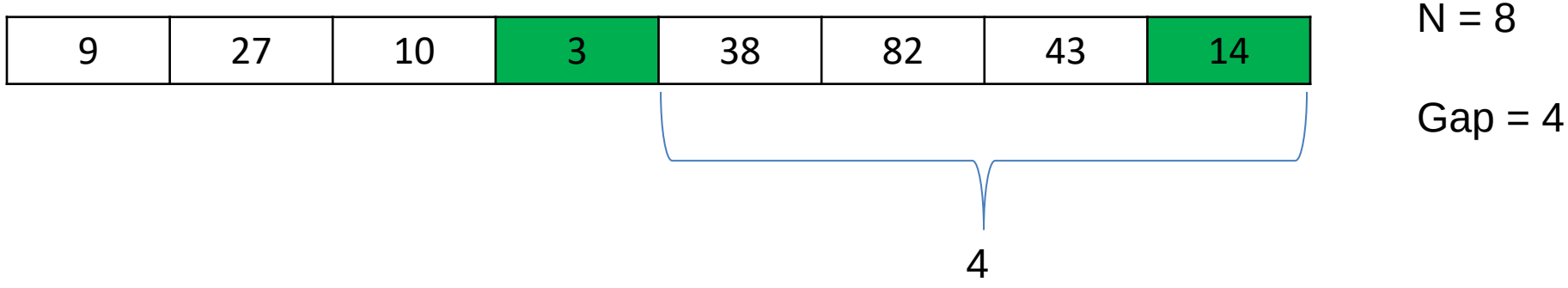


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

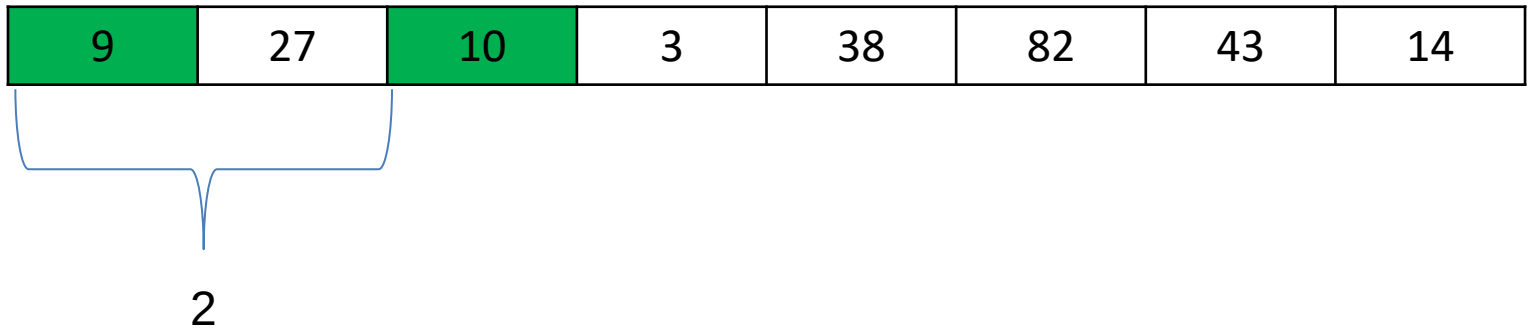


N = 8

Gap = 4

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



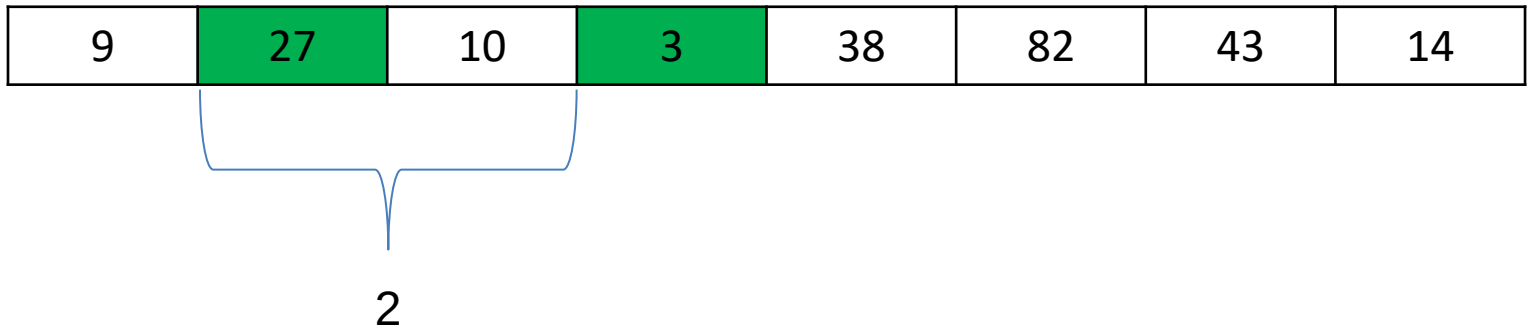
N = 8

~~Gap = 4~~

Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



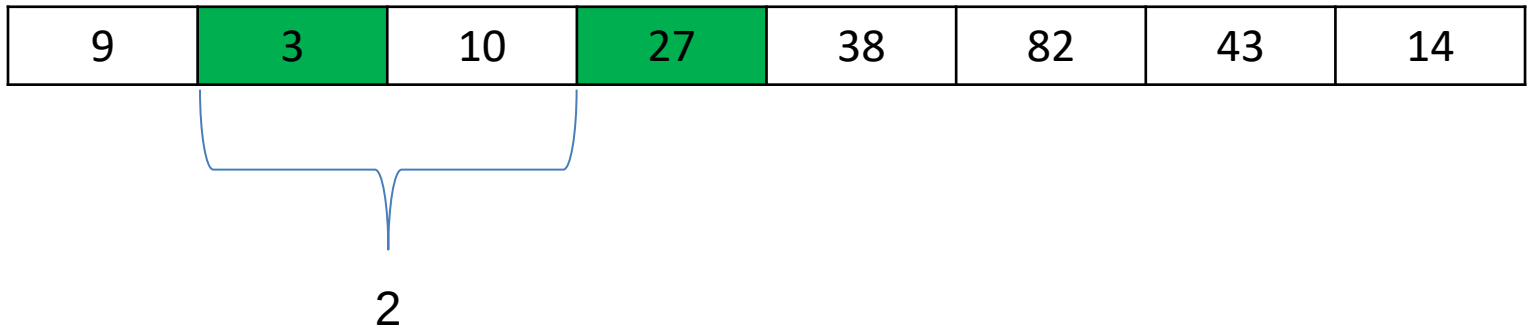
N = 8

~~Gap = 4~~

Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

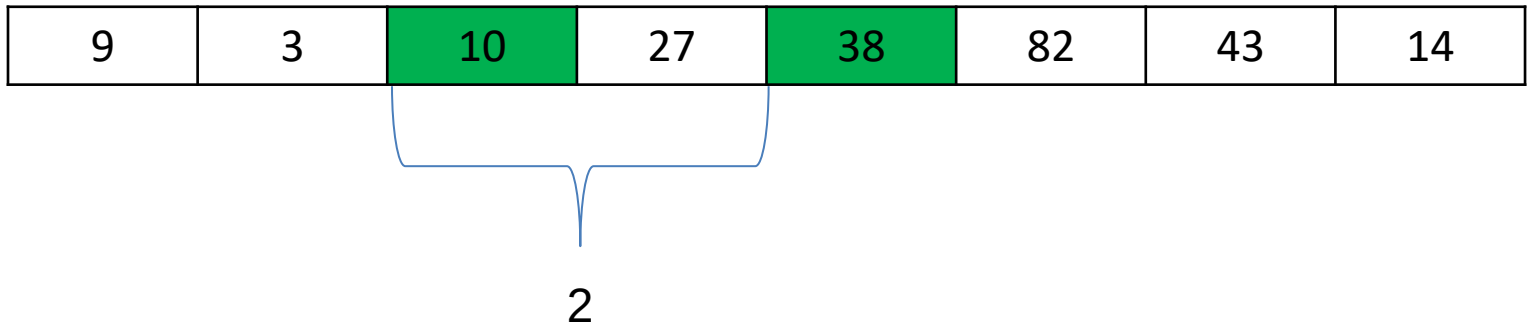


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

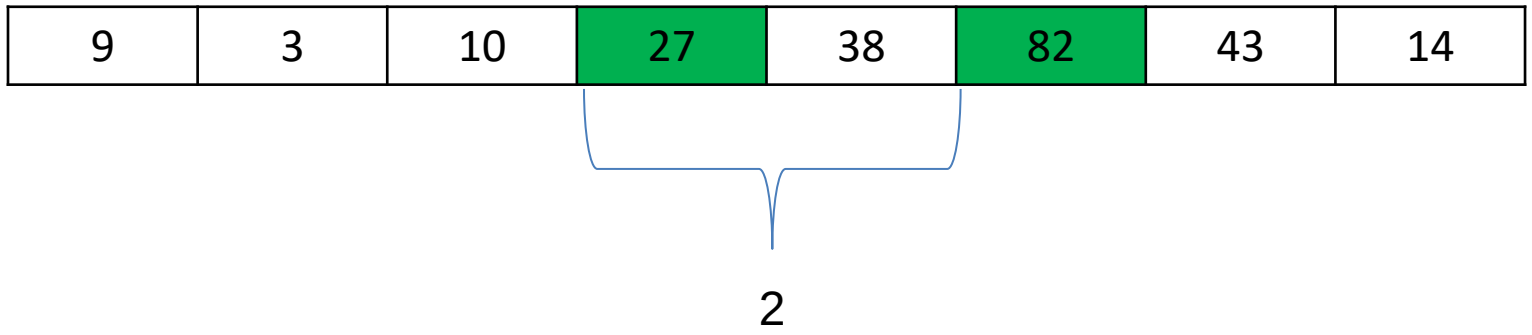


N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



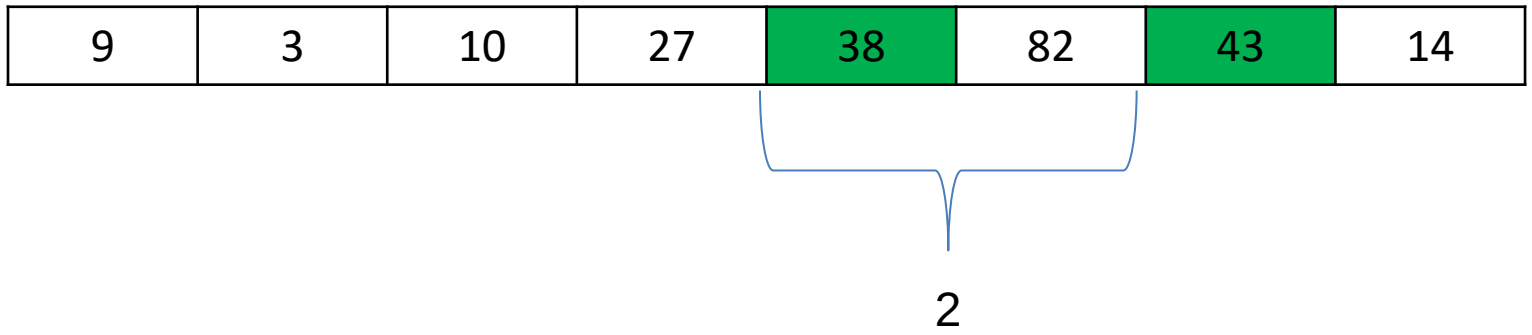
N = 8

~~Gap = 4~~

Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



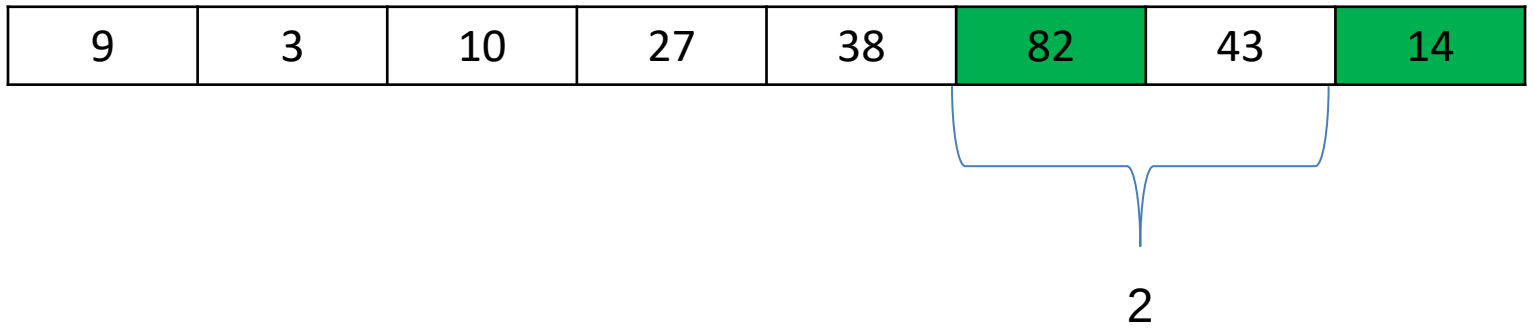
N = 8

~~Gap = 4~~

Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



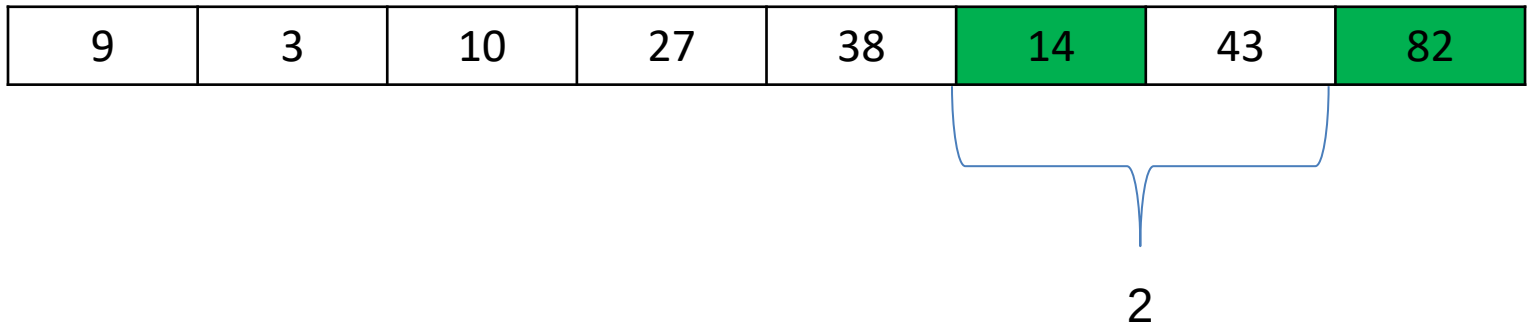
N = 8

~~Gap = 4~~

Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



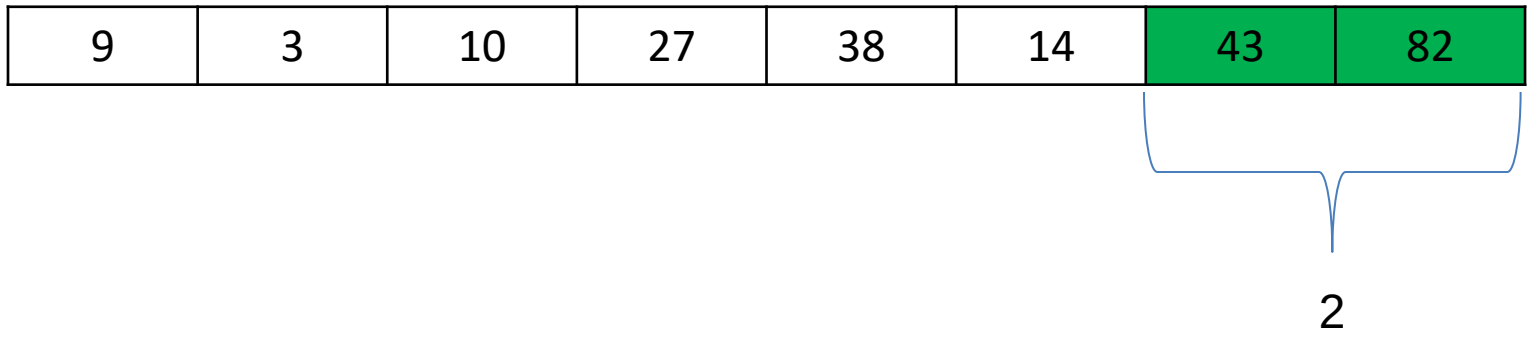
N = 8

~~Gap = 4~~

Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



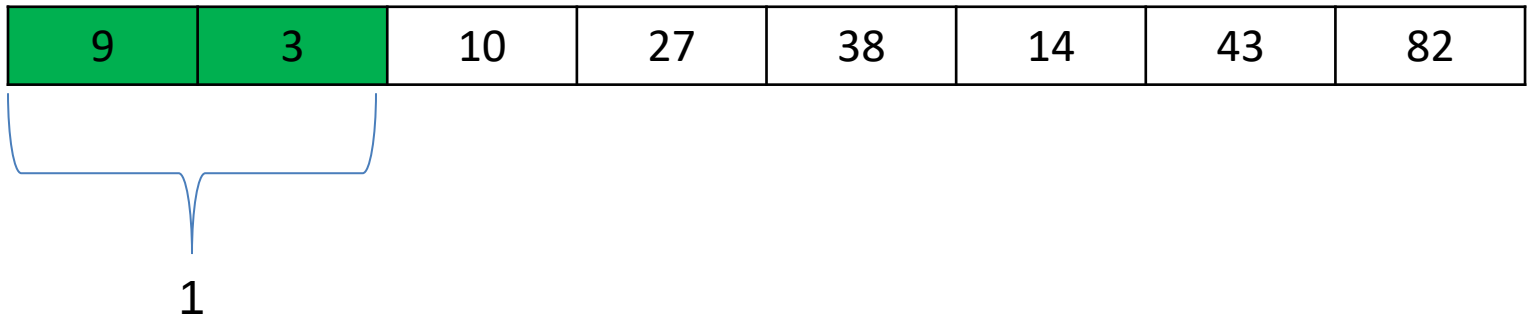
N = 8

~~Gap = 4~~

Gap = 2

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

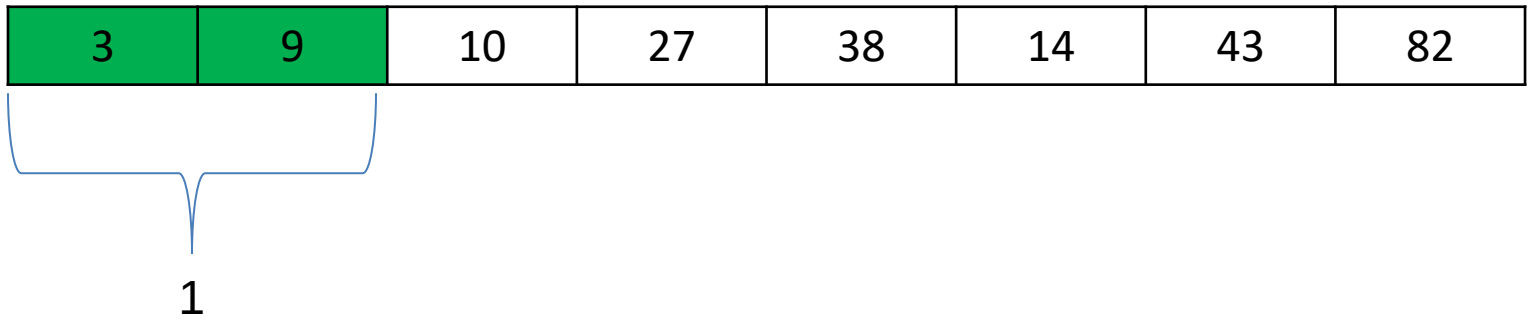


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

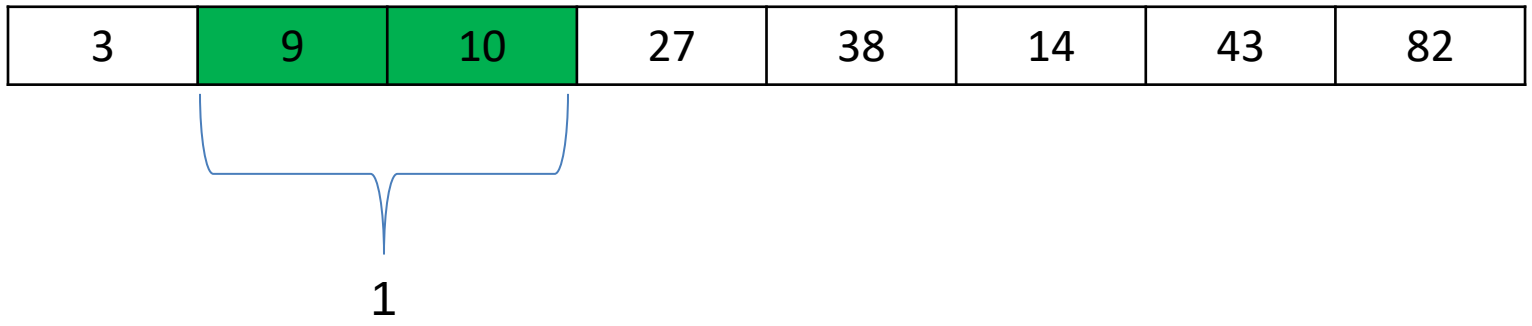


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

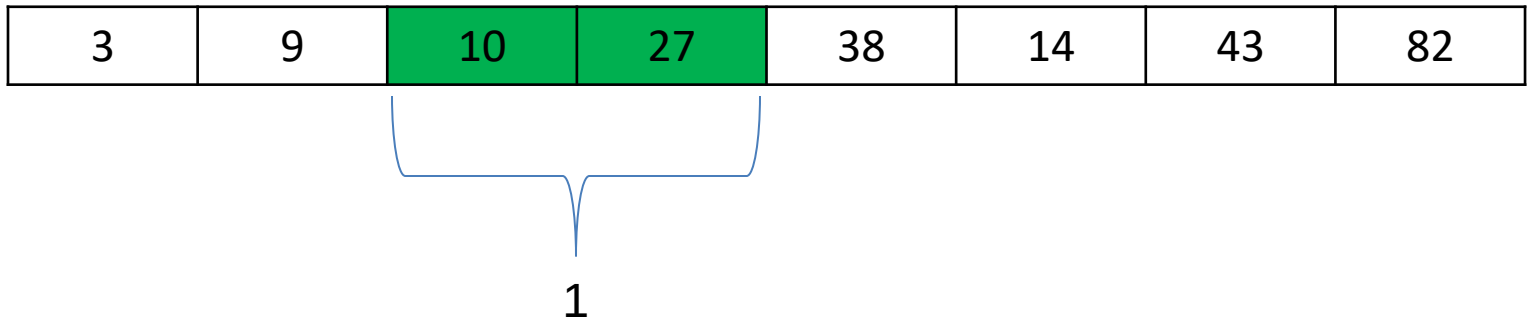


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

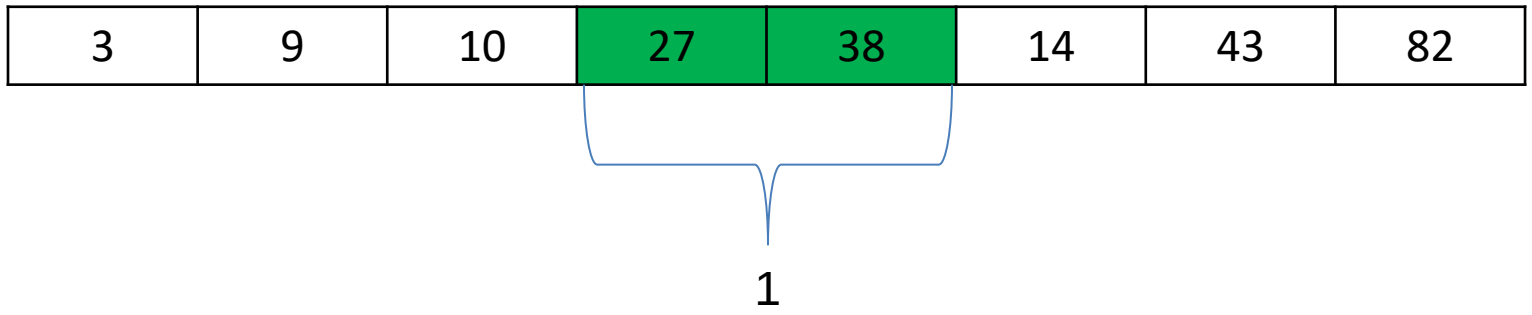


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

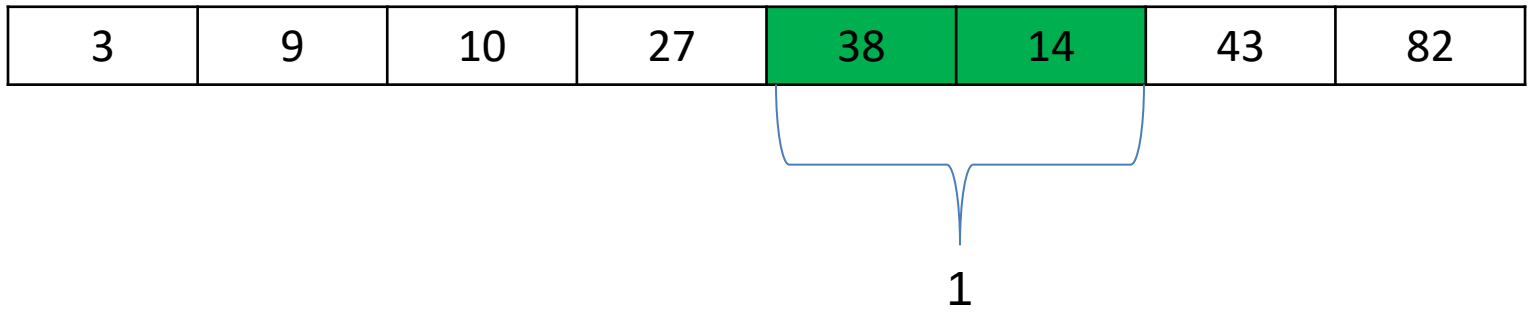


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

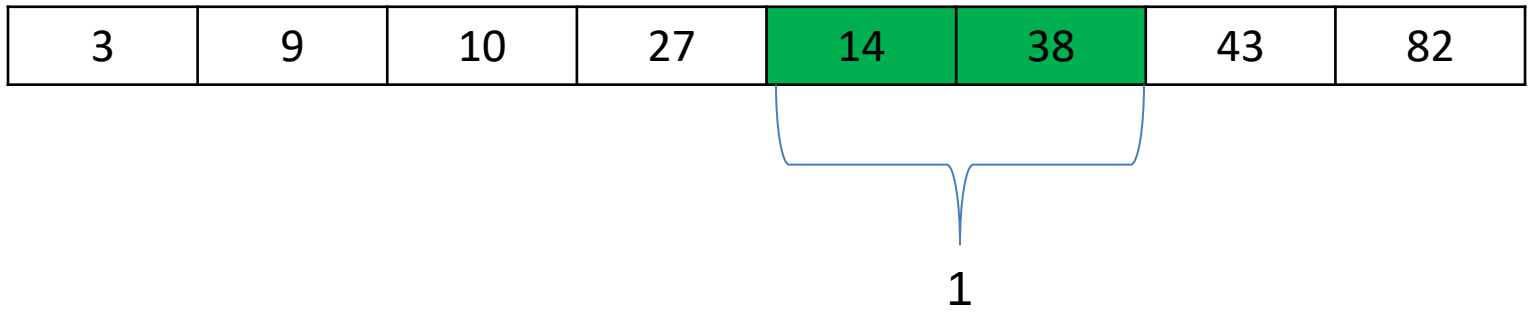


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

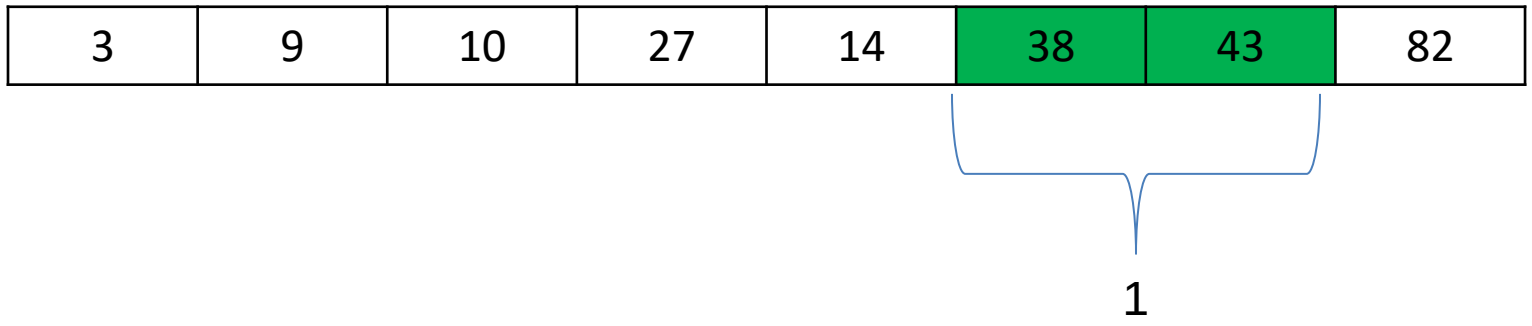


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



N = 8

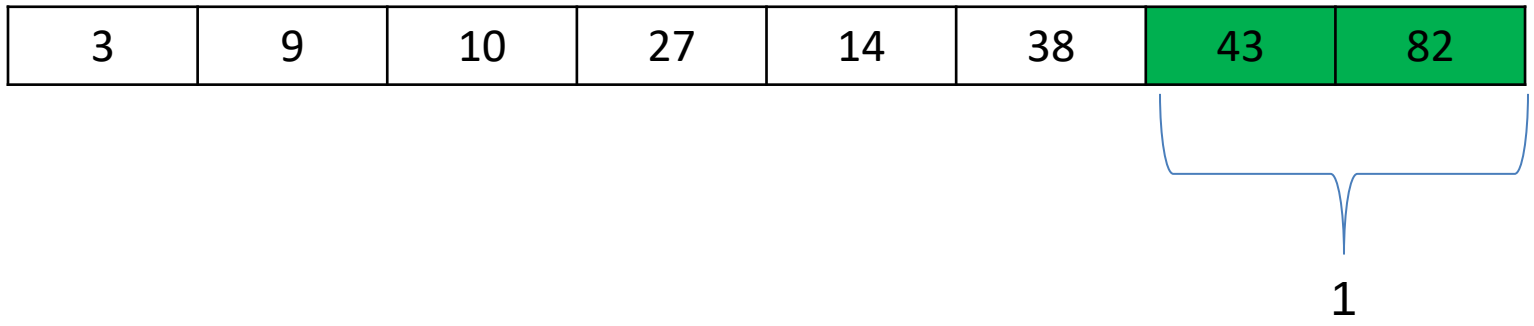
~~Gap = 4~~

~~Gap = 2~~

Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



N = 8

~~Gap = 4~~

~~Gap = 2~~

Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

3	9	10	27	14	38	43	82

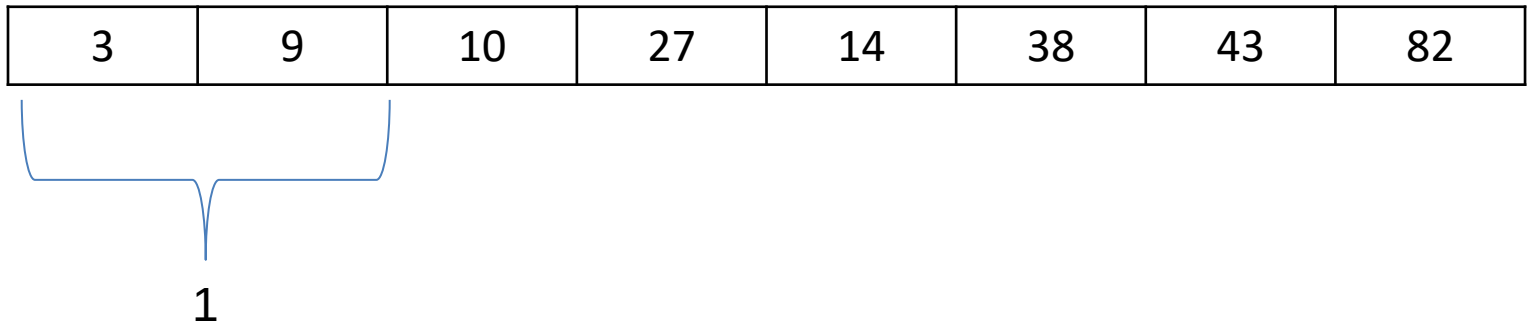
N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

(do it again ??)

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

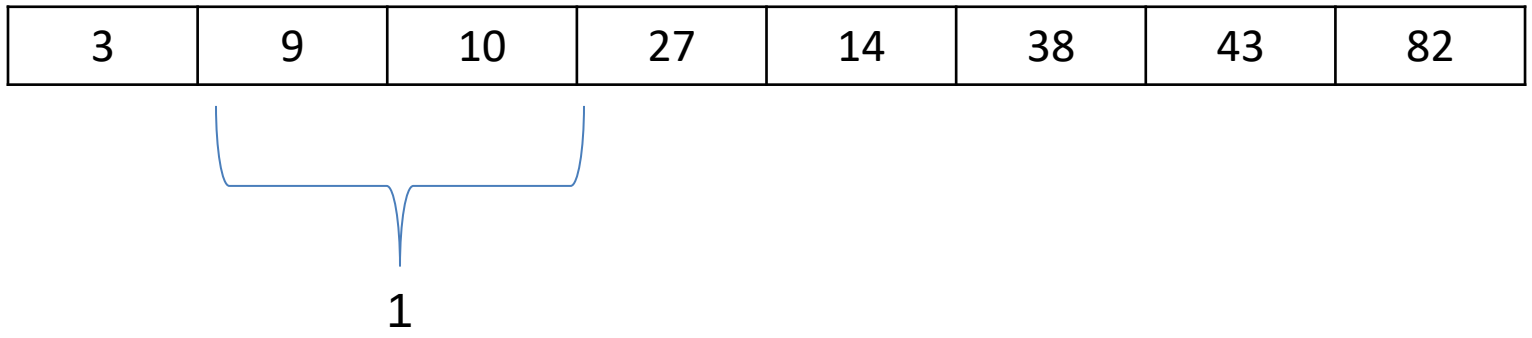


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

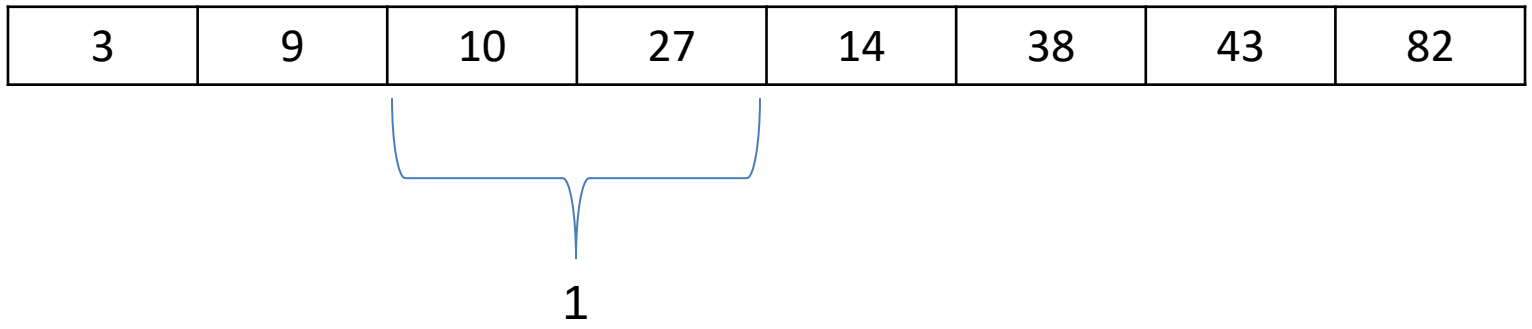


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

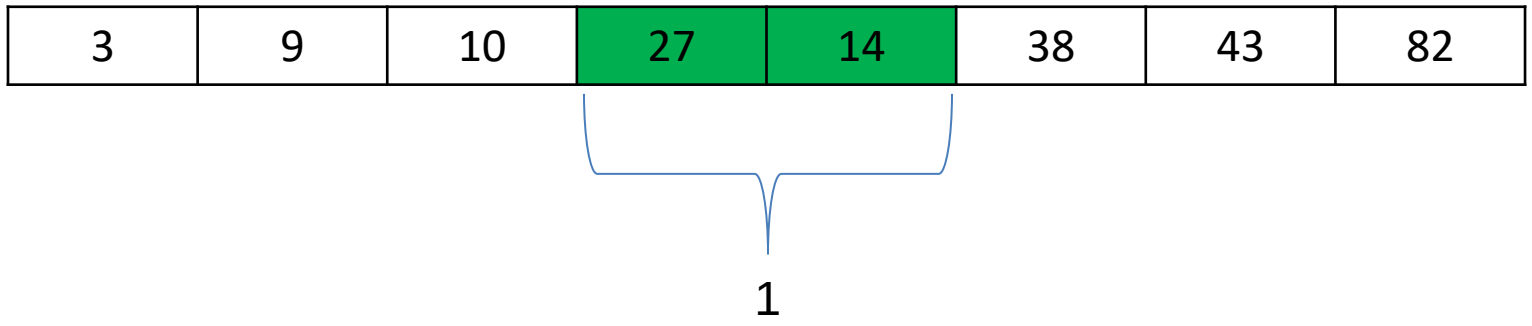


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

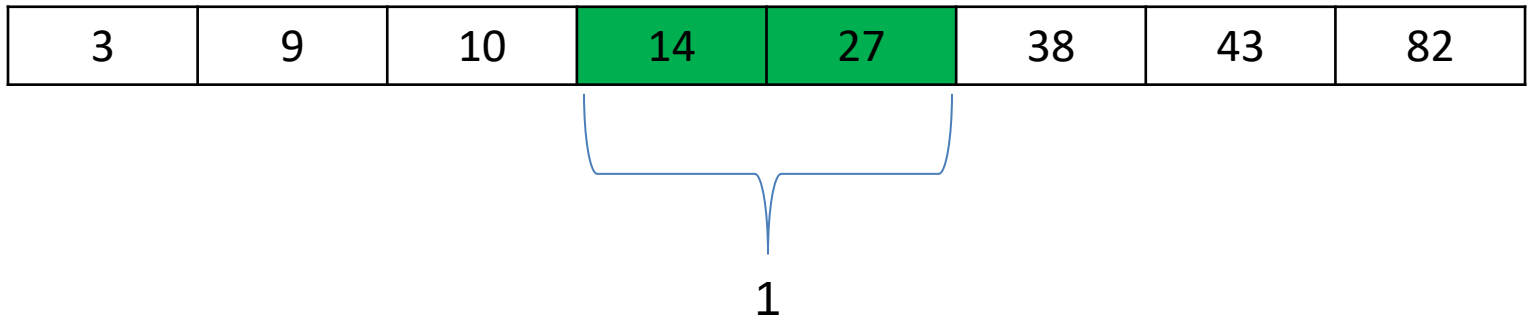


N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.



N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1

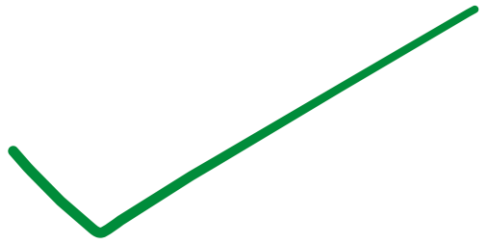
Shell Sort

Compare items that are distant from each other. After each iteration, decrease the gap size.

3	9	10	14	27	38	43	82

N = 8

~~Gap = 4~~
~~Gap = 2~~
Gap = 1



Running time: $O(n^2)$

Cocktail Shaker Sort

Double Sided Bubble Sort

https://en.wikipedia.org/wiki/Cocktail_shaker_sort

Running time: $O(n^2)$

Programming Languages Sorting Algorithms

Java has a built-in `sort` method for Arrays

What sorting algorithm does it use?

Method Detail

sort

```
public static void sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a - the array to be sorted`

https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

Programming Languages Sorting Algorithms

Java has a built-in `sort` method for Arrays

What sorting algorithm does it use?

Method Detail

sort

```
public static void sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

`a - the array to be sorted`

Python's `.sort()` function uses a hybrid of merge sort and insertion sort, called **Timsort**

Timsort

11 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

Timsort is a hybrid, stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It was implemented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently. This is done by merging runs until certain criteria are fulfilled. Timsort has been Python's standard sorting algorithm since version 2.3. It is also used to sort arrays of non-primitive type in Java SE 7,[4] on the Android platform,[5] in GNU Octave,[6] on V8,[7] Swift,[8] and Rust.[9]

It uses techniques from Peter McIlroy's 1993 paper "Optimistic Sorting and Information Theoretic Complexity".[10]

Timsort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$[1][2]
Best-case performance	$O(n)$[3]
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$

https://visualgo.net/en/sorting

Does anyone have any ideas for a very bad sorting algorithm, but still works?

Does anyone have any ideas for a very bad sorting algorithm, but still works?

If we are really lucky, our algorithm is insanely fast

If we are really unlucky, our algorithm will never finish

Bogo Sort (stupid sort) randomly shuffles the array until its sorted

```
while not sorted(array):
        shuffle(array)
```

Running time: O(pain) / undefined if we don't keep track of permutations checked

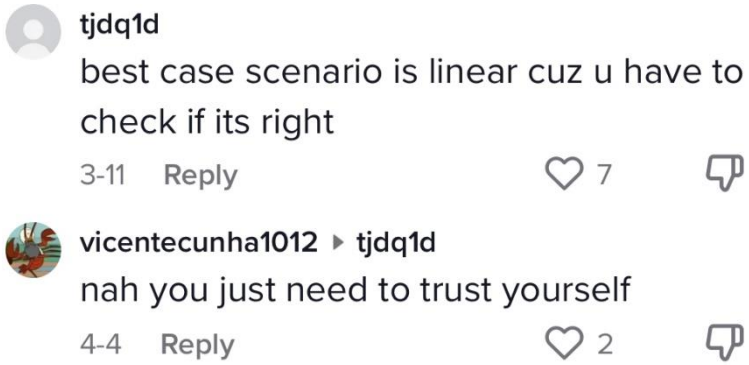
O(n!) if we keep track of permutations

Bogo Sort (stupid sort) randomly shuffles the array until its sorted

```
while not sorted(array):

        shuffle(array)
```

Best case scenario, this is the most efficient sorting algorithm!



Running time: O(pain) if we don't keep track of permutations checked

O(n!) if we keep track of permutations

This sorting algorithm is a joke, please don't take this one seriously…

Sorting Algorithms Visualized

https://youtu.be/kPRA0W1kECg