CSCI 132:

Basic Data Structures and Algorithms

Sorting (Part 4)

Reese Pearsall

Spring 2024

Announcements

Program 5 due Sunday May 5th

Lab 12 → Fill out the course evaluation

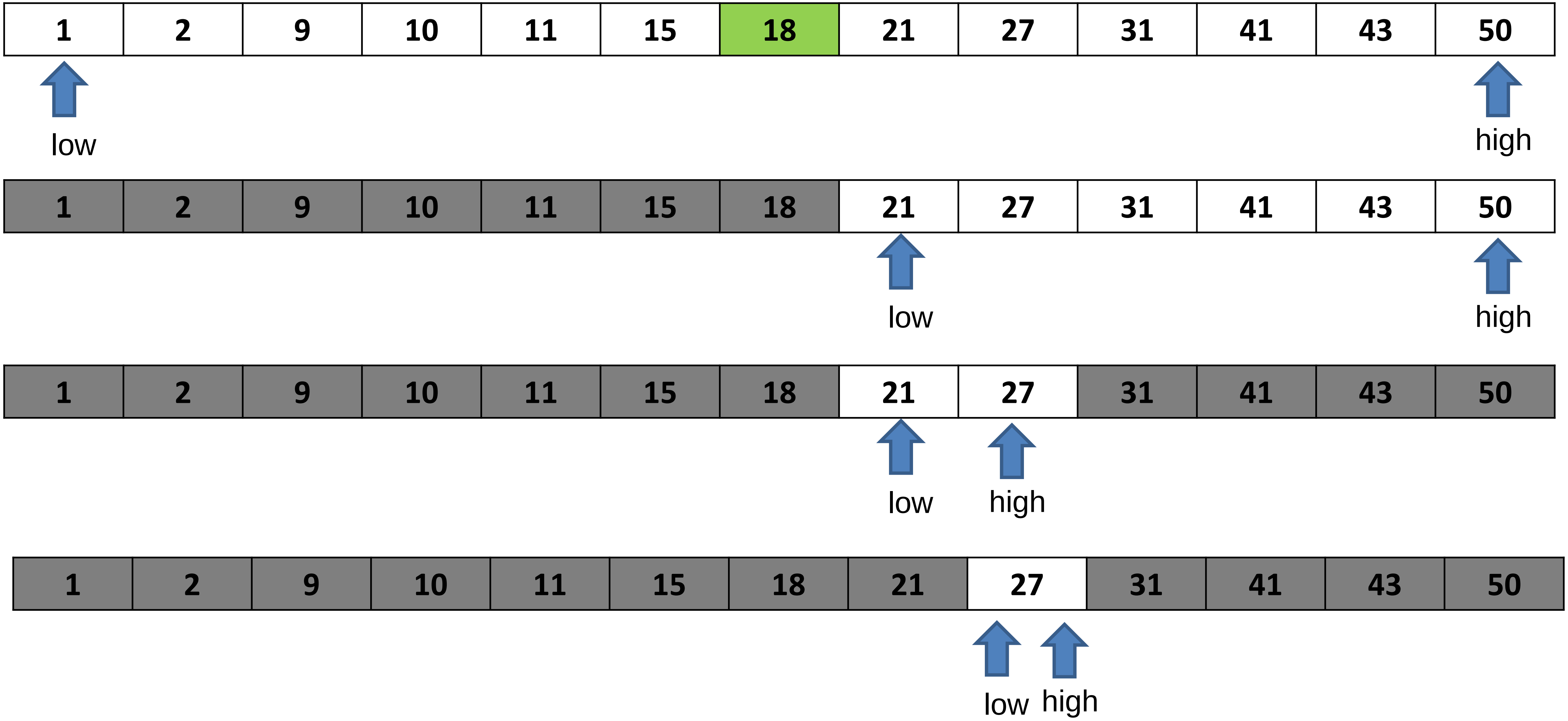
Rubber Duck Extra Credit Posted

Next Wednesday (5/1) is an optional help session for program 5 (no lecture)

```
private static int binary_search(int[] array, int n) {
      int low = 0; O(1)
      int high = array.length - 1; O(1)
      while(low <= high) { O(log n)
            int mid = (low + high) / 2;  O(1)
            if(n == array[mid]) { O(1)
                  return mid; O(1)
            }
            else if(n > array[mid]) { O(1)
                  low = mid + 1;  O(1)
            }
            else {
                  high = mid - 1; O(1)
            }
      }
      return -1; O(1)
}
```

Running time?   O(log n)

Why O(log n) ?



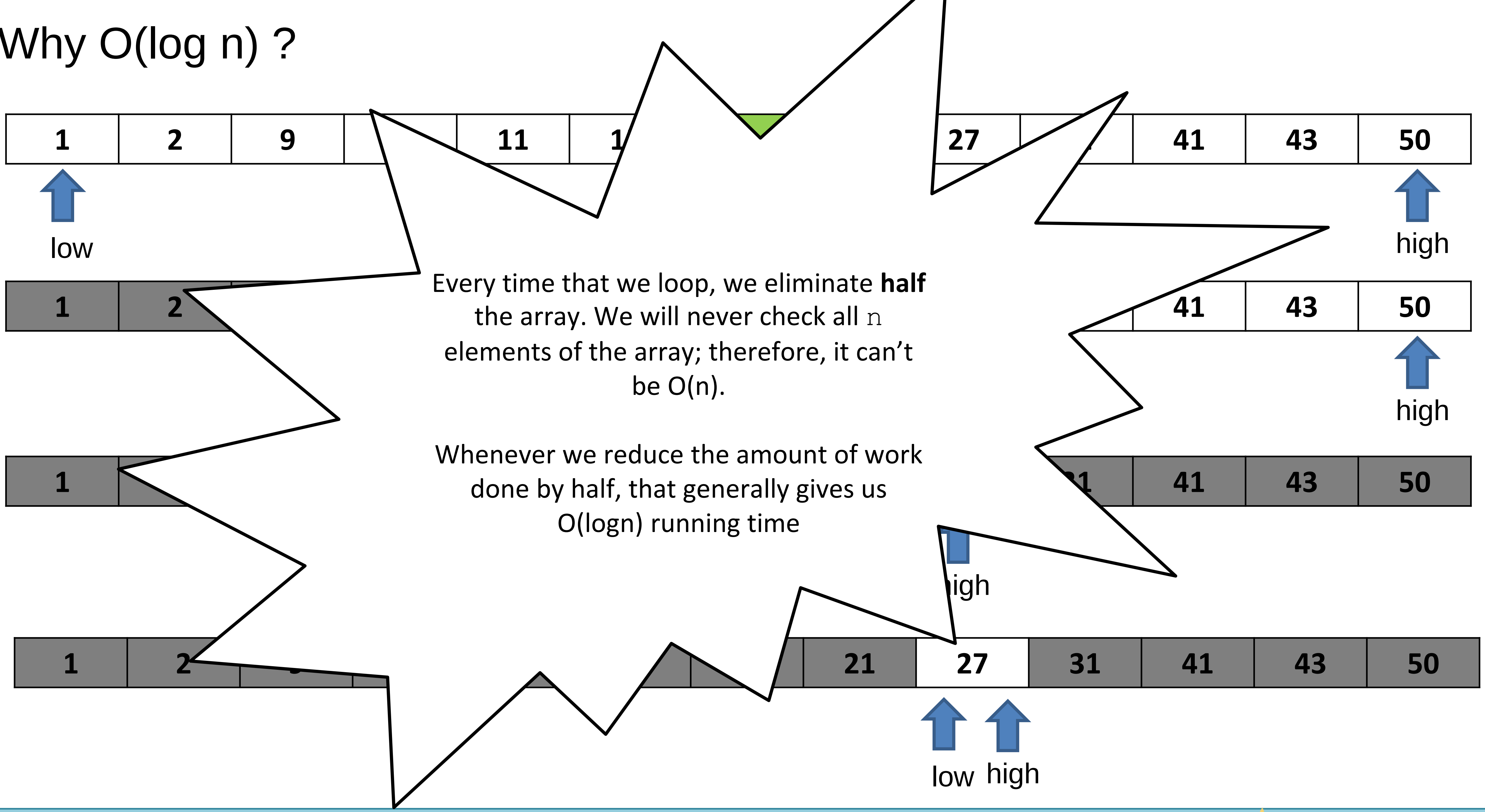
1	2	9	10	11	15	18	21	27	31	41	43	50

1	2	9	10	11	15	18	21	27	31	41	43	50

1	2	9	10	11	15	18	21	27	31	41	43	50

1	2	9	10	11	15	18	21	27	31	41	43	50

Why O(log n) ?



Every time that we loop, we eliminate **half** the array. We will never check all `n` elements of the array; therefore, it can’t be O(n).

Whenever we reduce the amount of work done by half, that generally gives us O(logn) running time

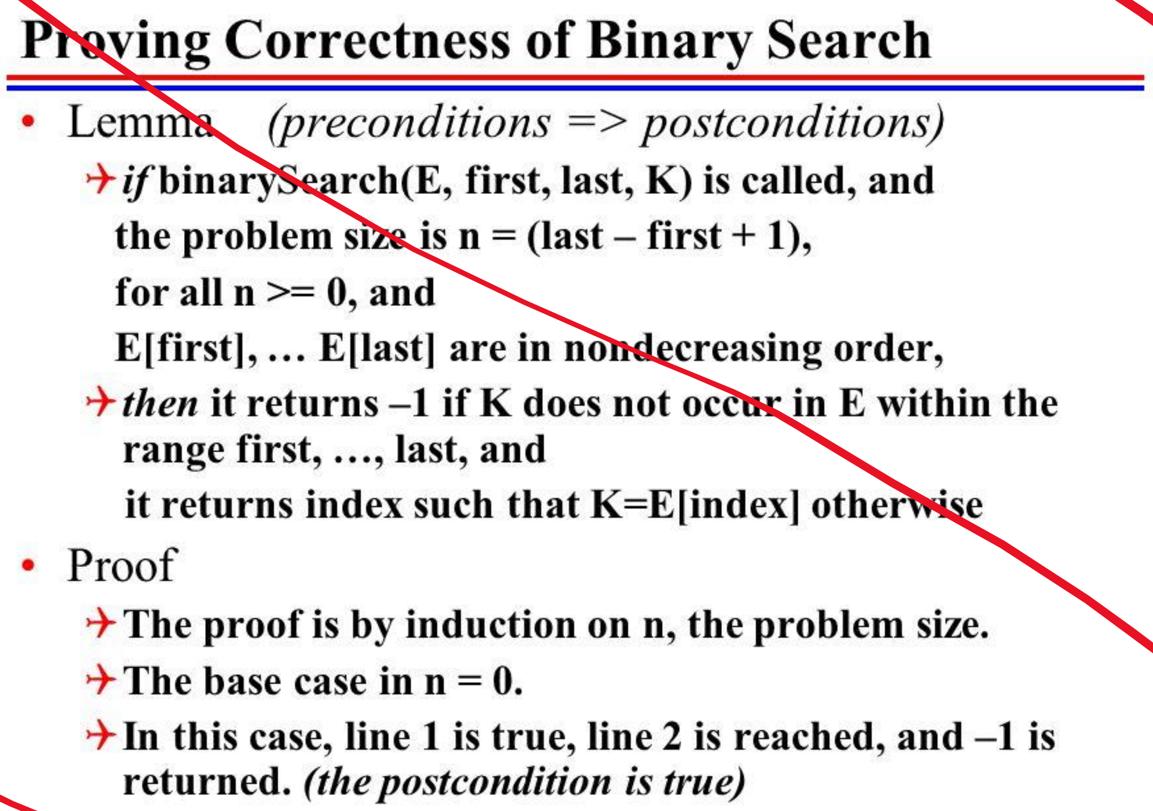
```
private static int binary_search(?????????????) {

    if(low <= high) {
        int mid = (low + high) / 2;
        if(n == array[mid]) {
            return mid;
        }
        else if(n > array[mid]) {
            return binary_search(?????????);
        }
        else {
            return binary_search(?????????);
        }
    }
    else {
        return -1;
    }
}
```

Binary Search can also be implemented using recursion

```java
private static int binary_search_recursive(int[] array, int n, int high, int low) {
        if(low <= high) {
                int mid = (low + high) / 2;
                if(n == array[mid]) {
                        return mid;
                }
                else if(n > array[mid]) {
                        return binary_search_recursive(array, n, high, mid+1);
                }
                else {
                        return binary_search_recursive(array, n, mid-1, low);
                }
        }
        else {
                return -1;
        }
}
```

Binary Search can also be implemented using recursion



Proving Correctness of Binary Search

- Lemma *(preconditions => postconditions)*
  - *if* binarySearch(E, first, last, K) is called, and
    the problem size is n = (last – first + 1),
    for all n >= 0, and
    E[first], … E[last] are in nondecreasing order,
  - *then* it returns –1 if K does not occur in E within the range first, …, last, and
    it returns index such that K=E[index] otherwise
- Proof
  - The proof is by induction on n, the problem size.
  - The base case in n = 0.
  - In this case, line 1 is true, line 2 is reached, and –1 is returned. *(the postcondition is true)*

Running Time of Sorting Algorithms

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

```java
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;
}
```

You will not be tested about today’s sorting algorithms.

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

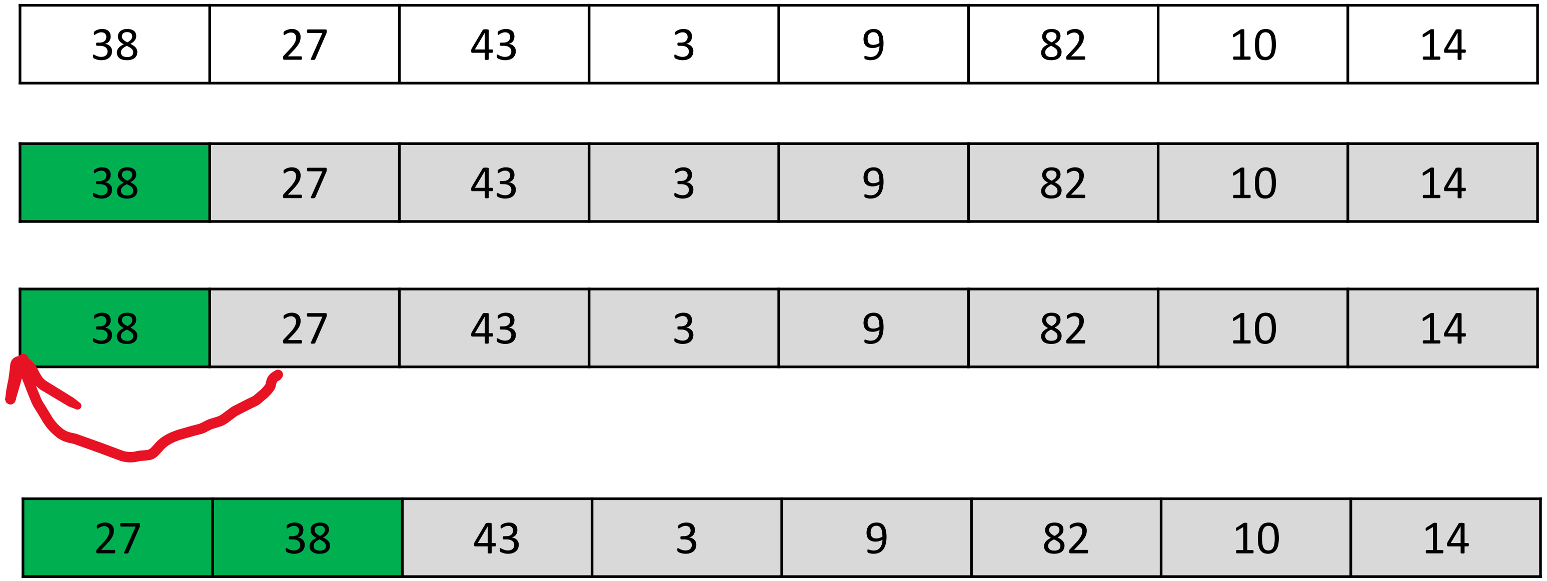
We divide our array into two sections. A sorted section, and an unsorted section. We iterate through the array, and for each iteration, we move one element from the unsorted section to the sorted section

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



38	27	43	3	9	82	10	14

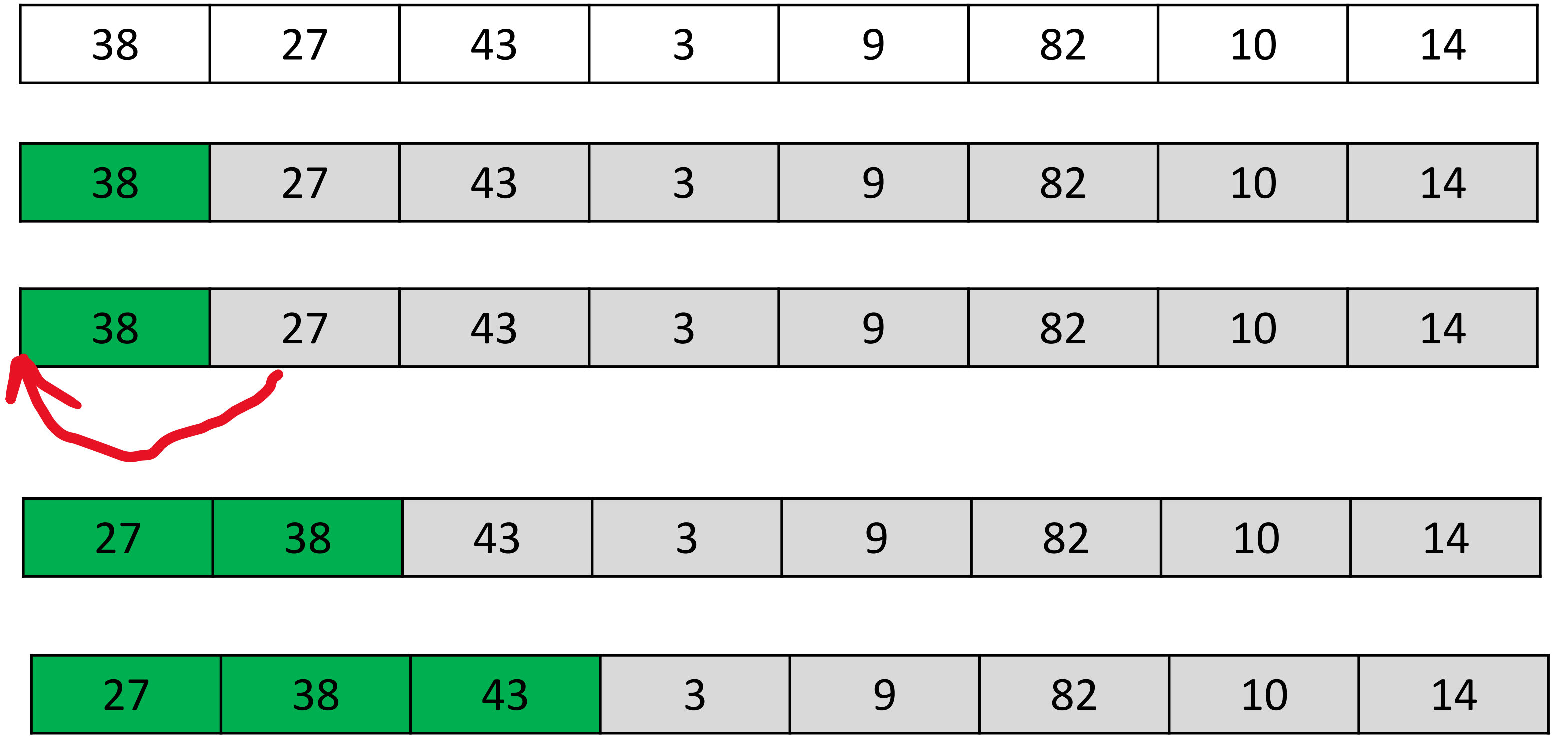
38	27	43	3	9	82	10	14

38	27	43	3	9	82	10	14

27	38	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

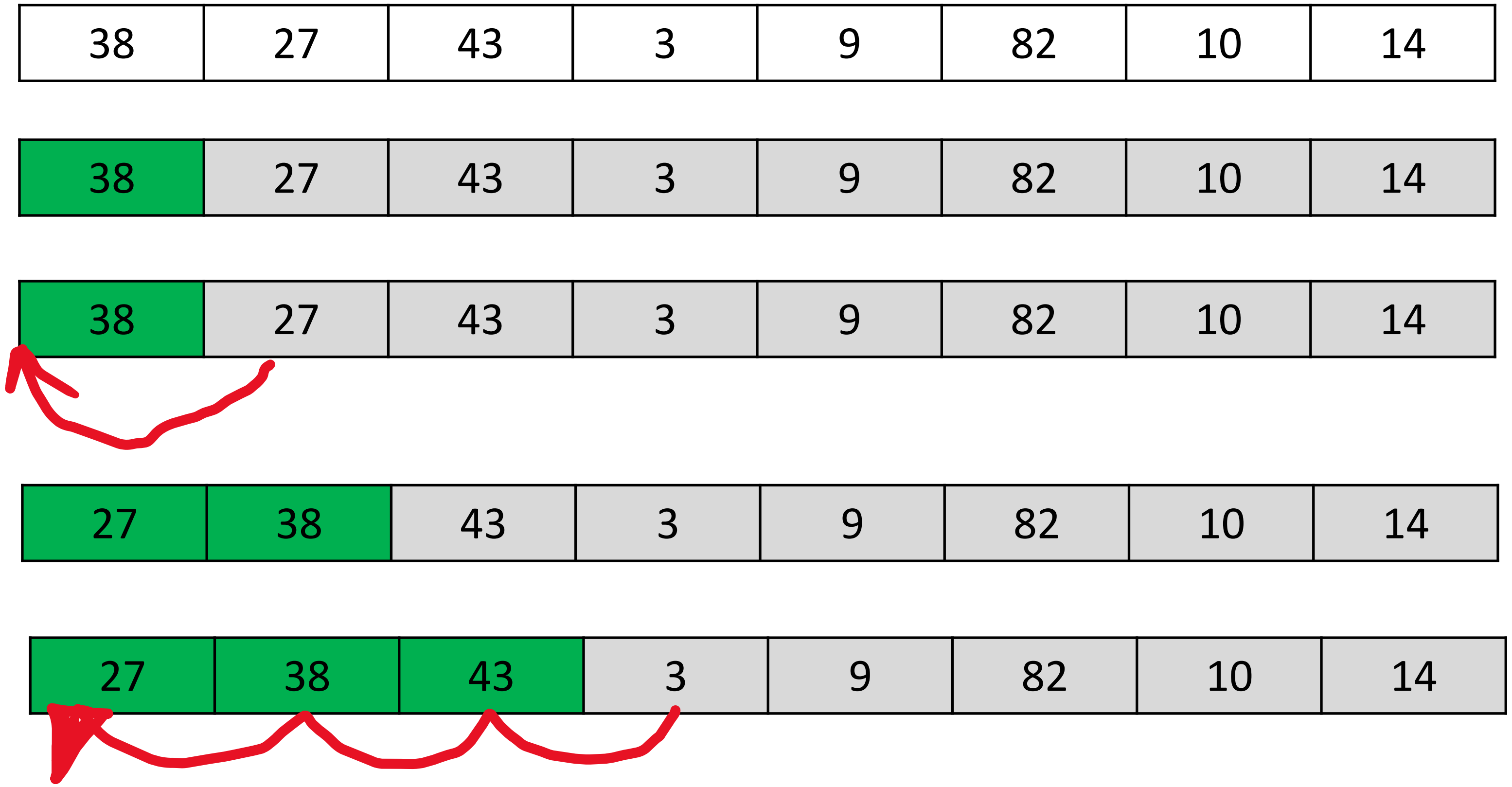
38	27	43	3	9	82	10	14

27	38	43	3	9	82	10	14

27	38	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

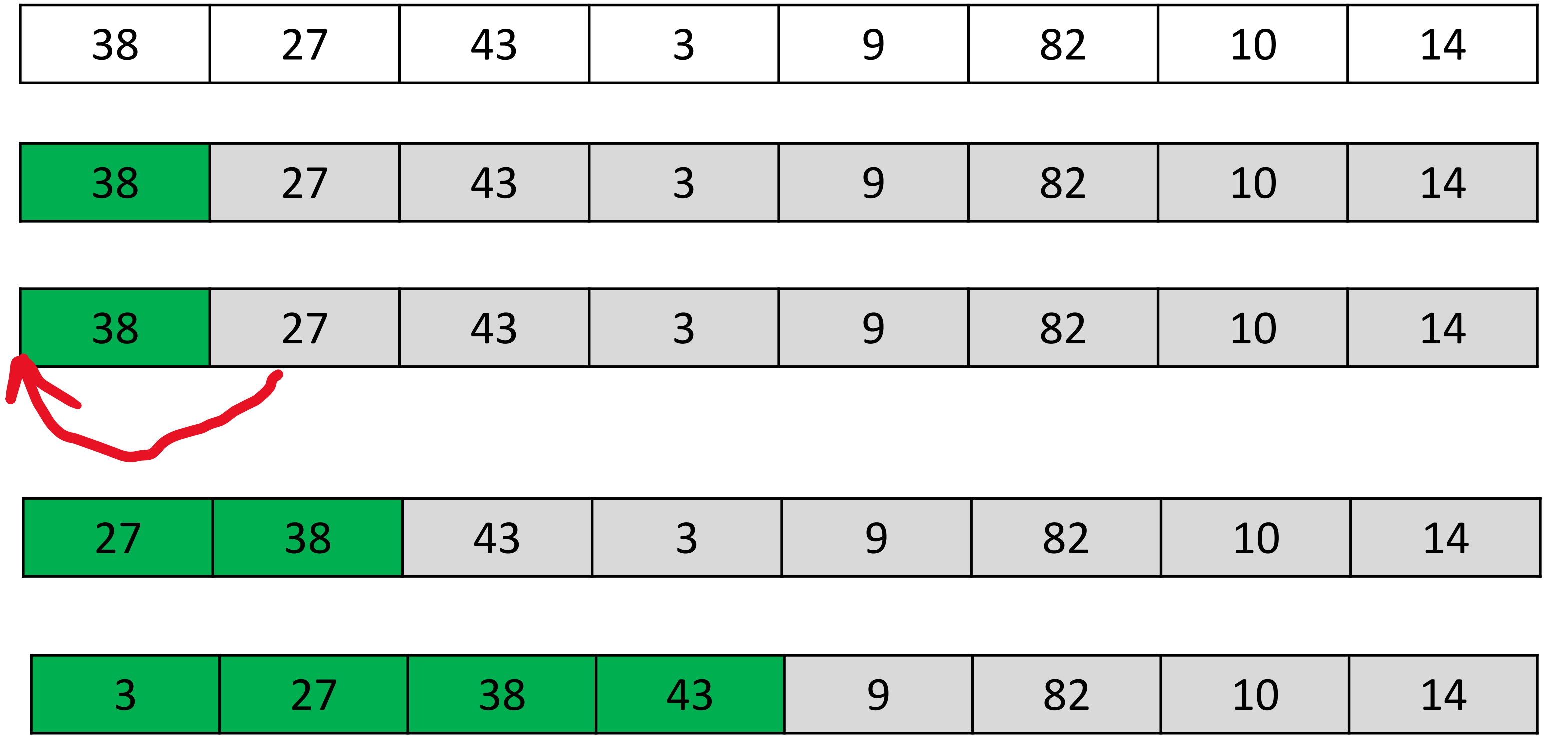
38	27	43	3	9	82	10	14

27	38	43	3	9	82	10	14

27	38	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

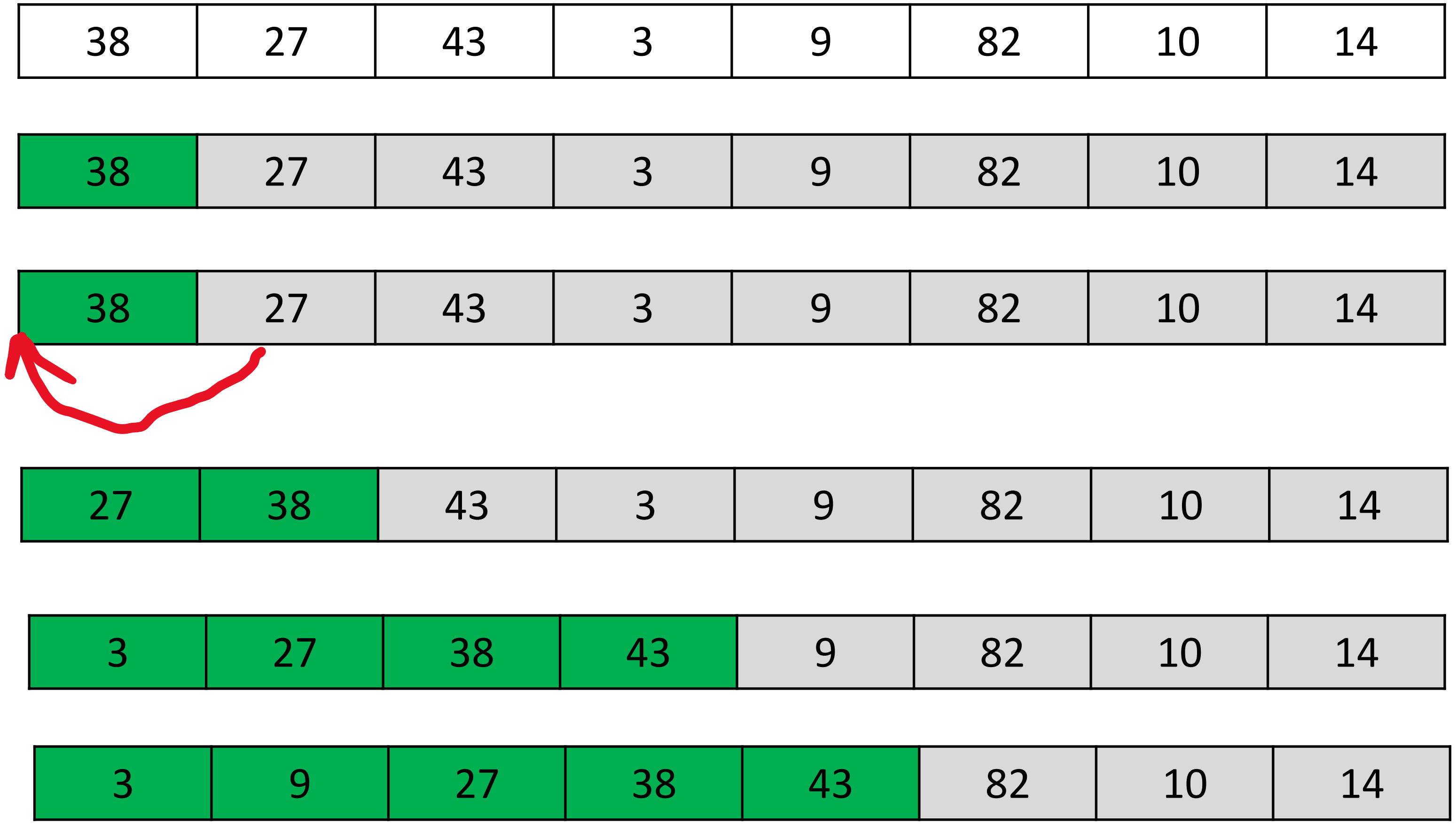
38	27	43	3	9	82	10	14

27	38	43	3	9	82	10	14

3	27	38	43	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

38 27 43 3 9 82 10 14

27 38 43 3 9 82 10 14

3 27 38 43 9 82 10 14

3 9 27 38 43 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

3	9	27	38	43	82	10	14

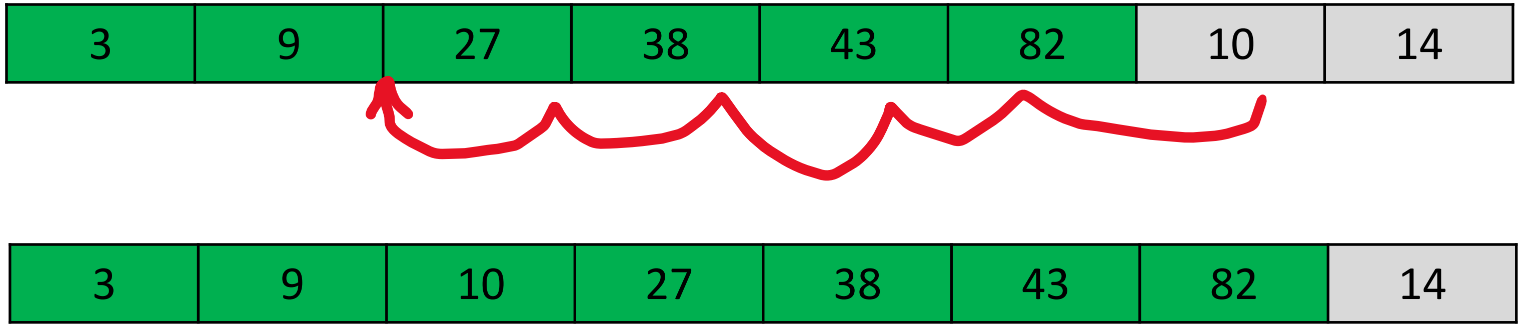
Insertion Sort

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

3	9	27	38	43	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

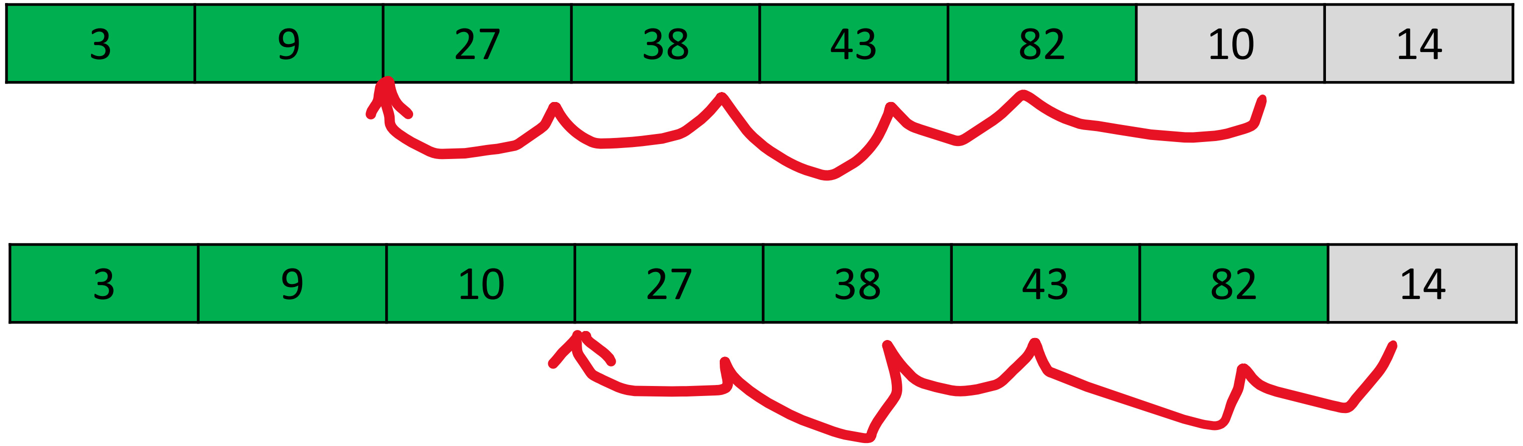


3	9	27	38	43	82	10	14

3	9	10	27	38	43	82	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

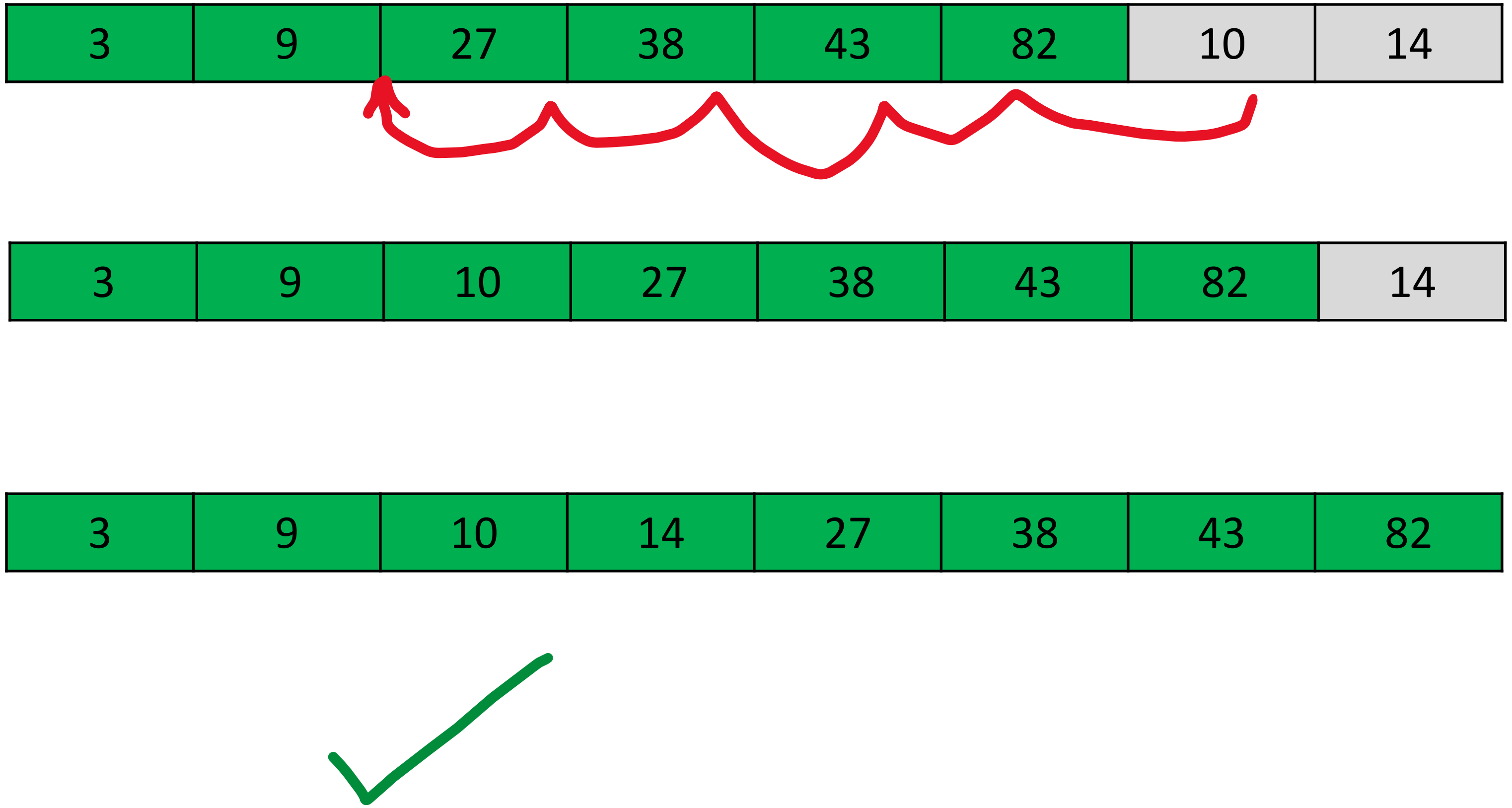


3	9	27	38	43	82	10	14

3	9	10	27	38	43	82	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



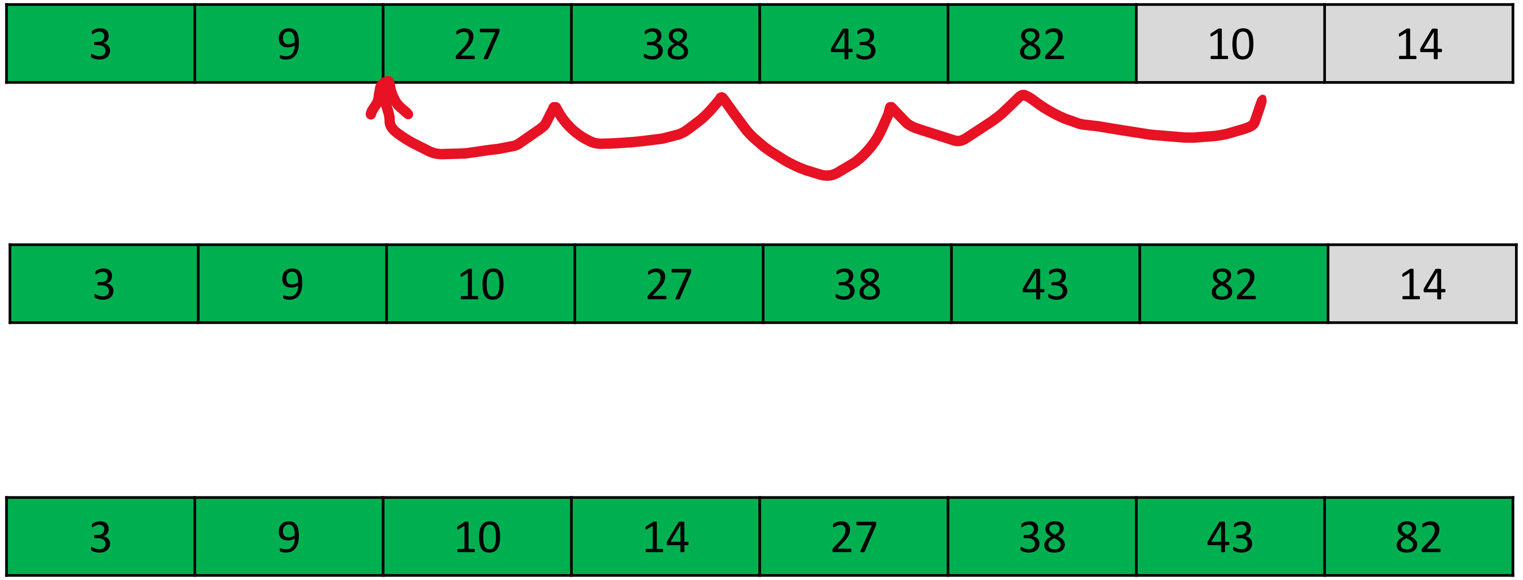
3	9	27	38	43	82	10	14

3	9	10	27	38	43	82	14

3	9	10	14	27	38	43	82

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



3	9	27	38	43	82	10	14

3	9	10	27	38	43	82	14

3	9	10	14	27	38	43	82

✓

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.

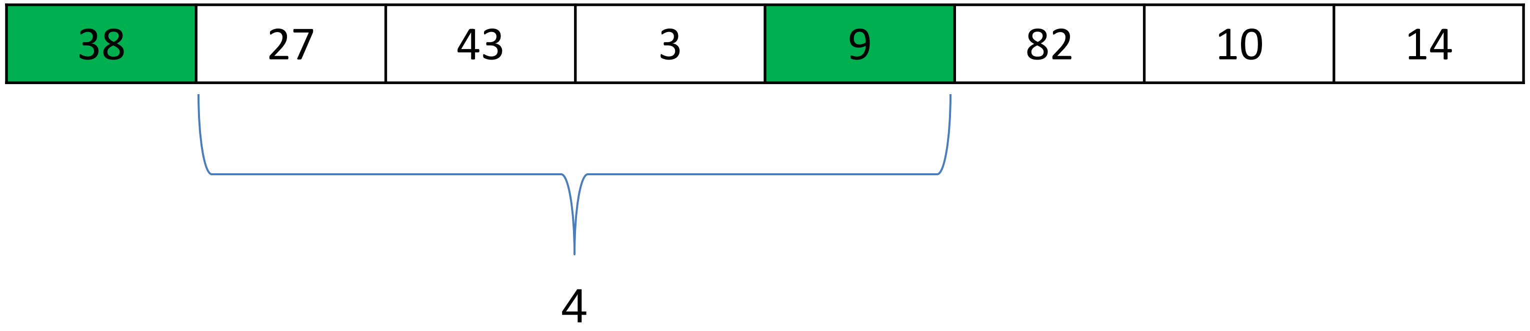
38	27	43	3	9	82	10	14

N = 8

Gap = 4

Shell Sort

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



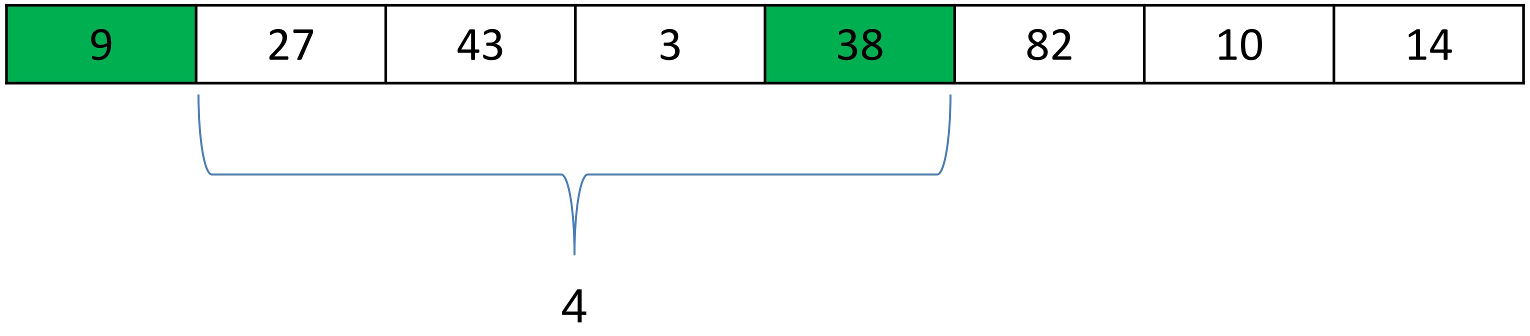
38	27	43	3	9	82	10	14

N = 8

Gap = 4

Shell Sort

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



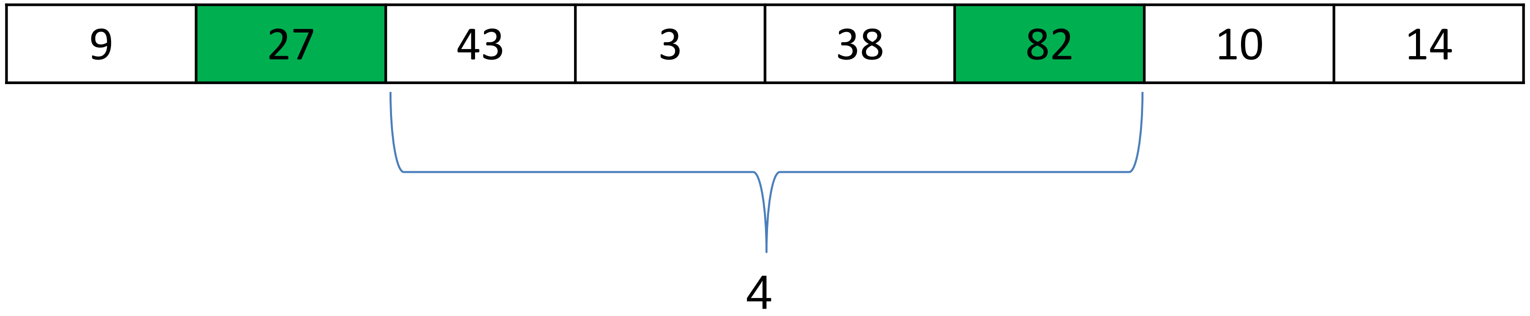
9	27	43	3	38	82	10	14

N = 8

Gap = 4

Shell Sort

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



9	27	43	3	38	82	10	14

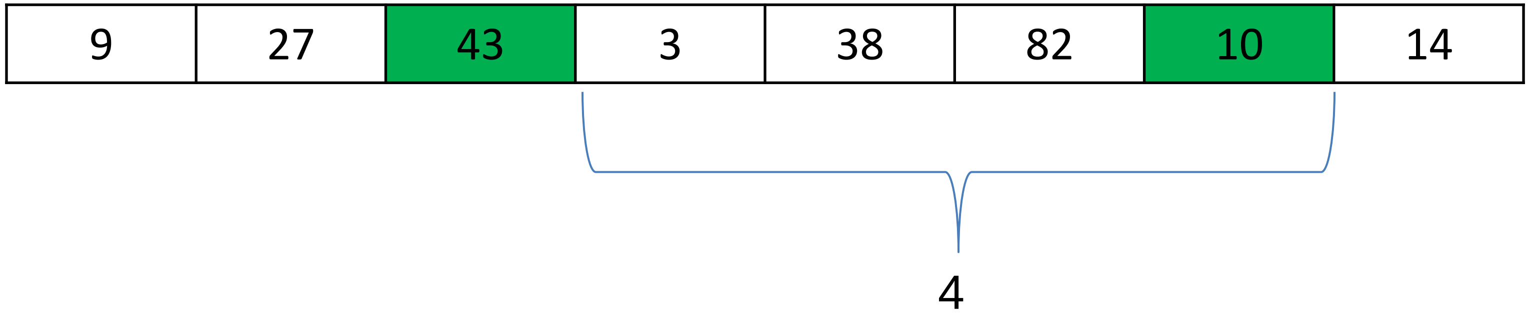
4

N = 8

Gap = 4

Shell Sort

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



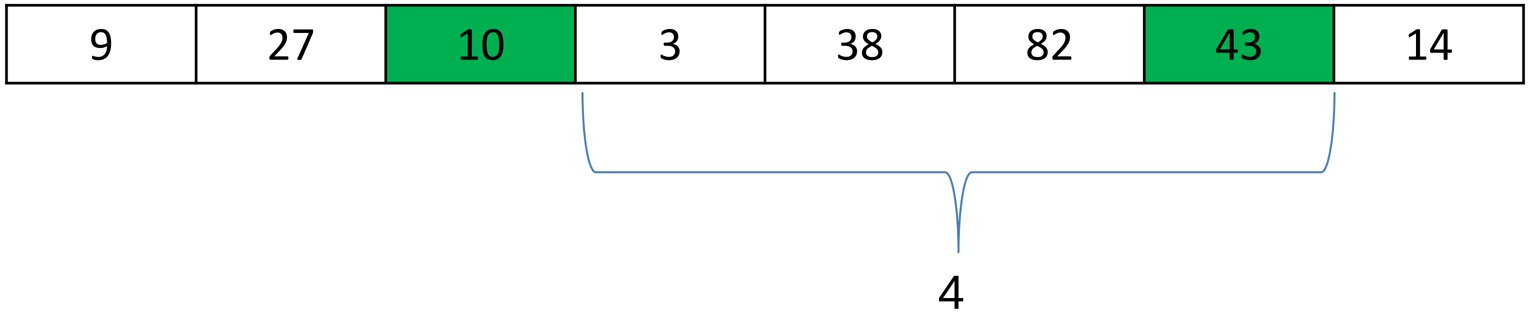
9	27	43	3	38	82	10	14

N = 8

Gap = 4

Shell Sort

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



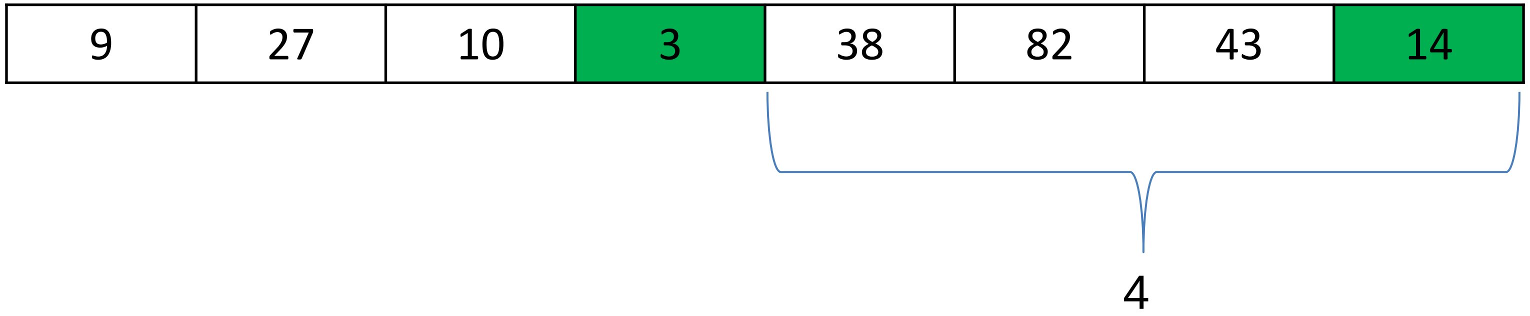
9	27	10	3	38	82	43	14

N = 8

Gap = 4

Shell Sort

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



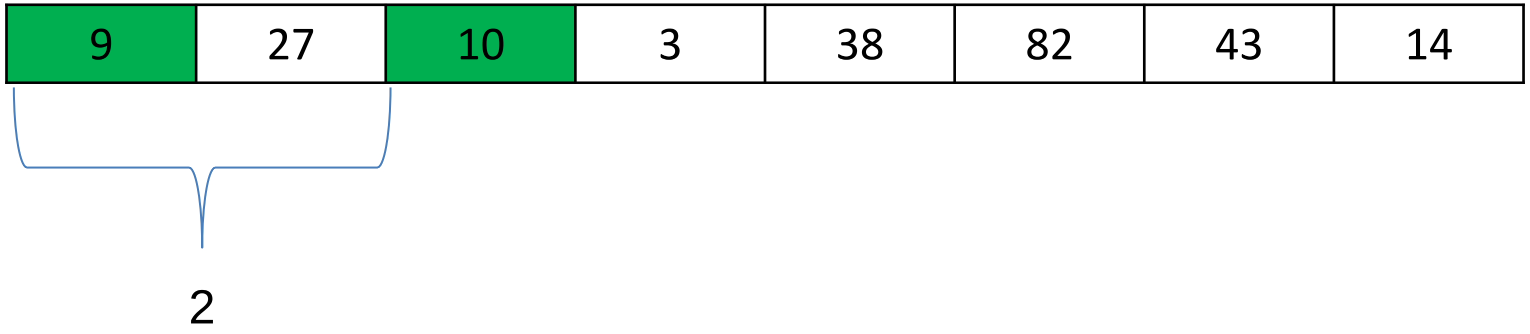
9	27	10	3	38	82	43	14

N = 8

Gap = 4

Shell Sort

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



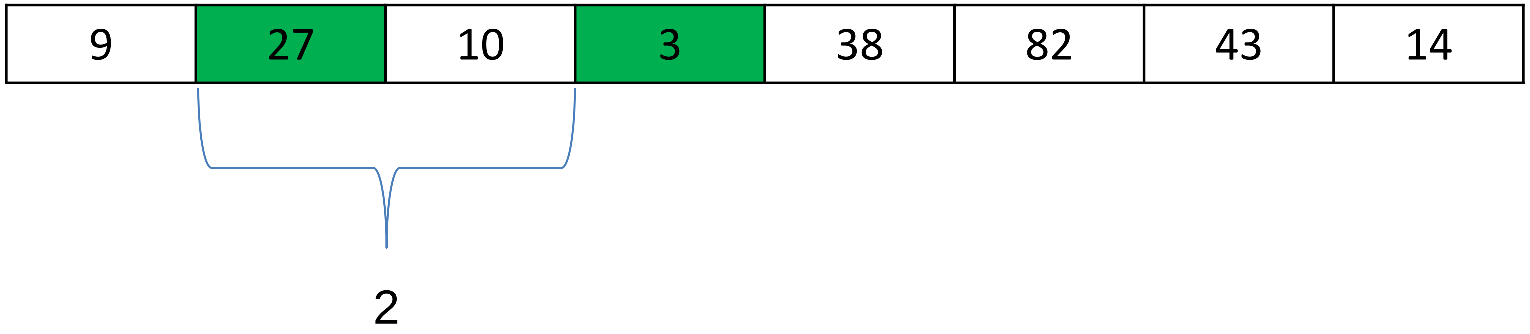
9	27	10	3	38	82	43	14

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



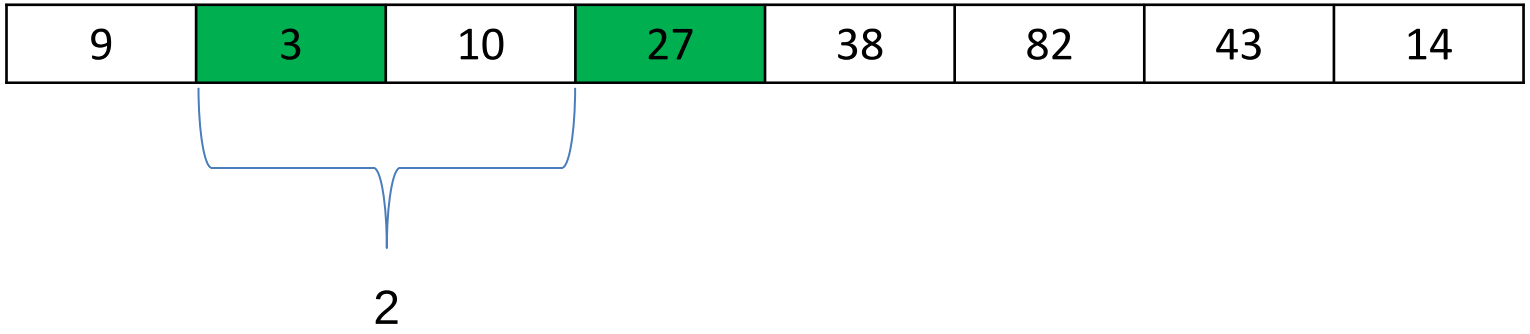
9	27	10	3	38	82	43	14

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



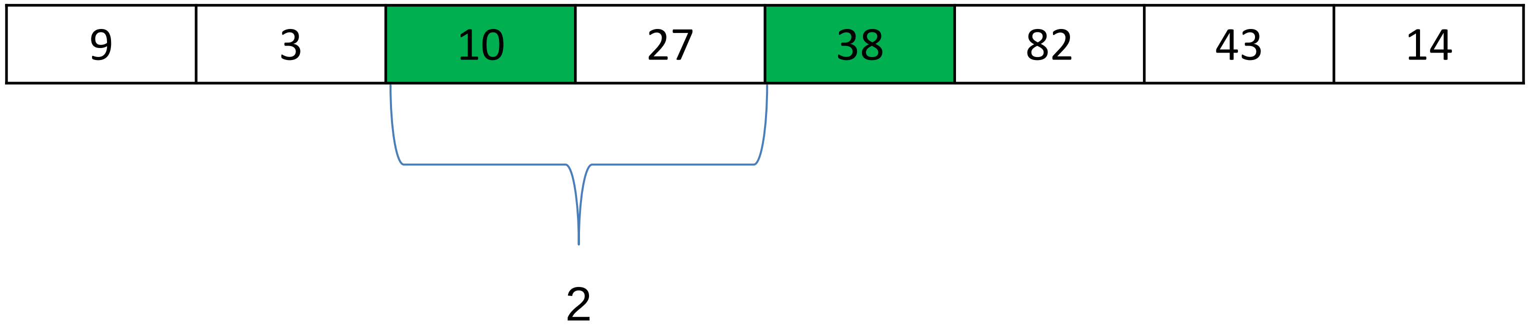
9	3	10	27	38	82	43	14

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



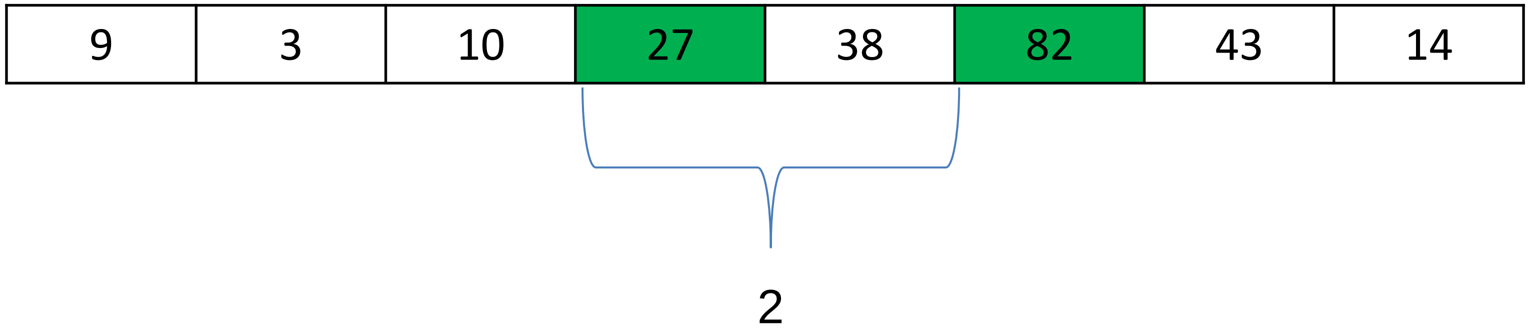
9	3	10	27	38	82	43	14

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



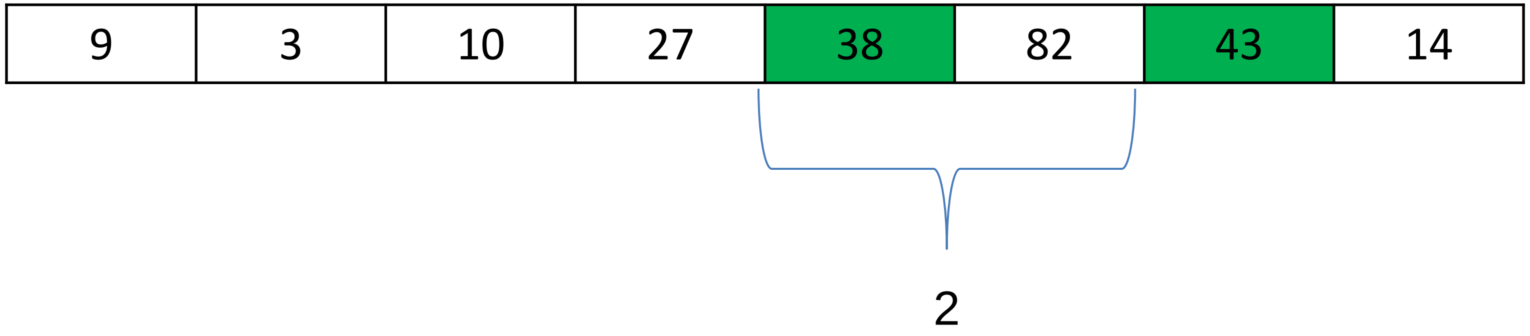
9	3	10	27	38	82	43	14

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



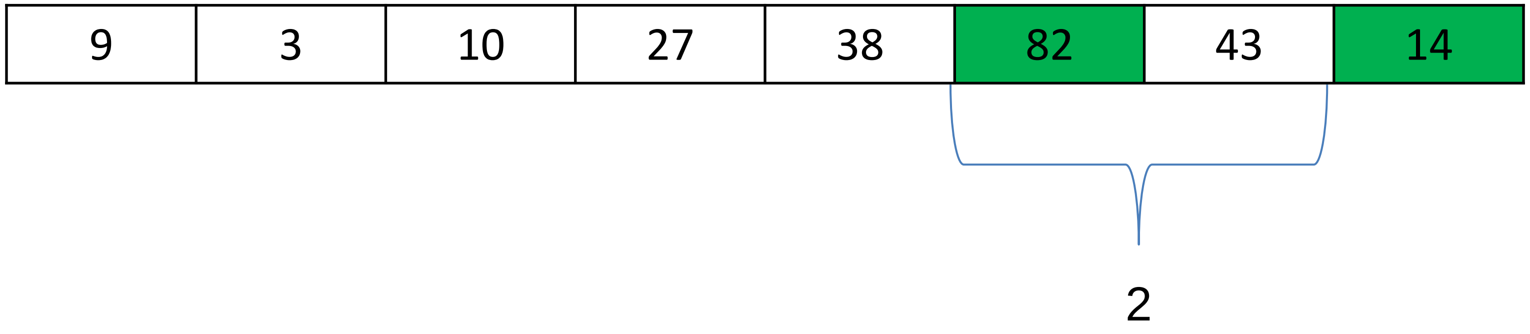
9	3	10	27	38	82	43	14

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



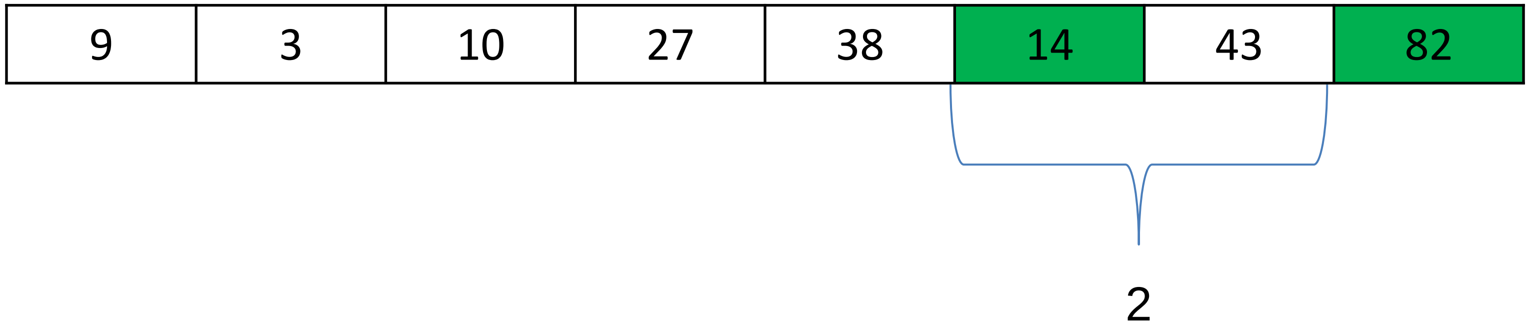
9	3	10	27	38	82	43	14

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



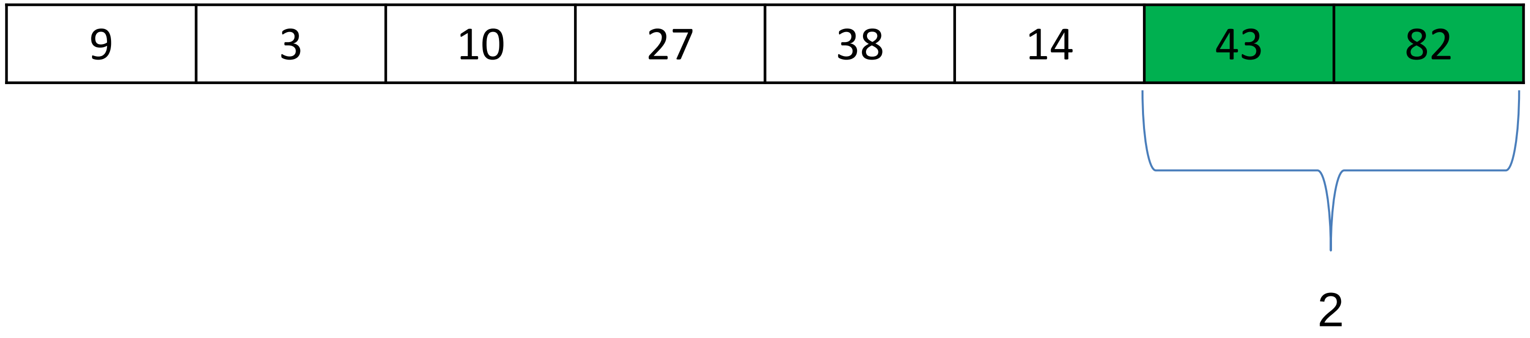
9	3	10	27	38	14	43	82

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



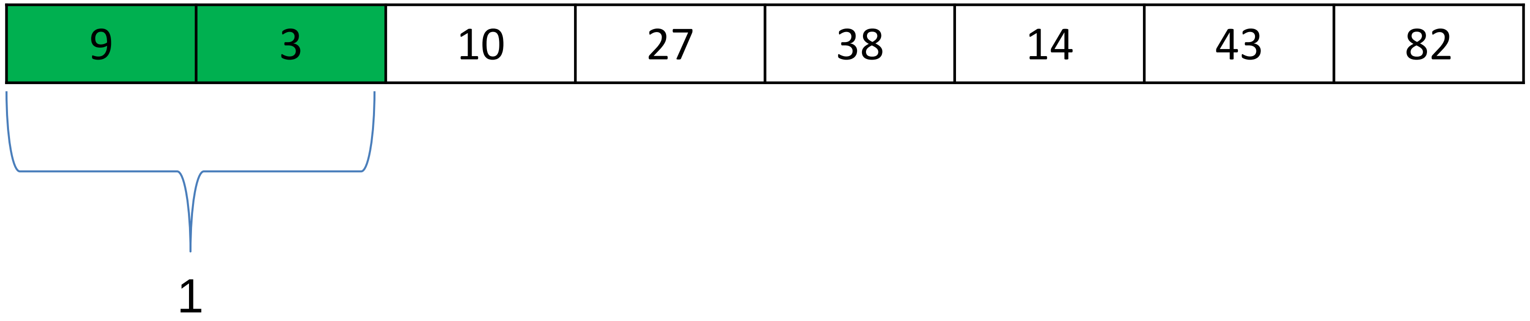
9	3	10	27	38	14	43	82

N = 8

~~Gap = 4~~
Gap = 2

Shell Sort

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



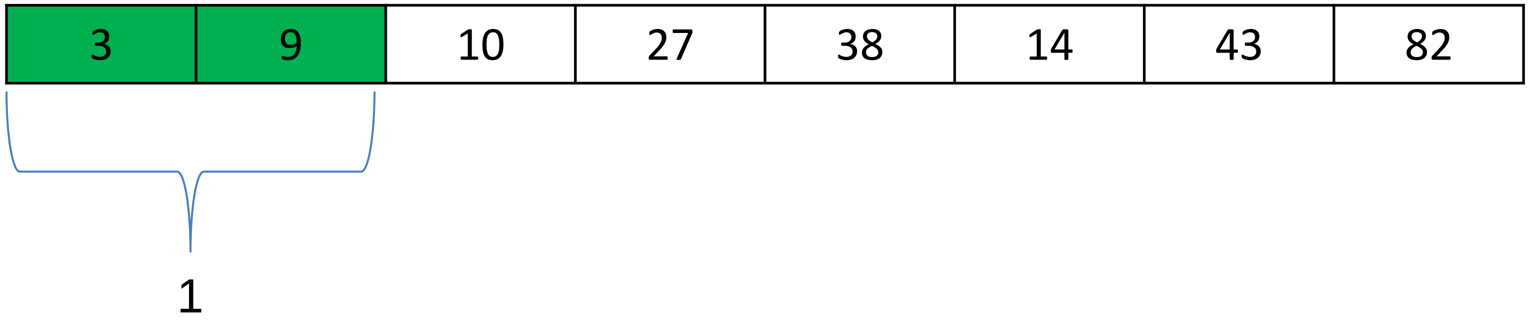
9	3	10	27	38	14	43	82

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	38	14	43	82

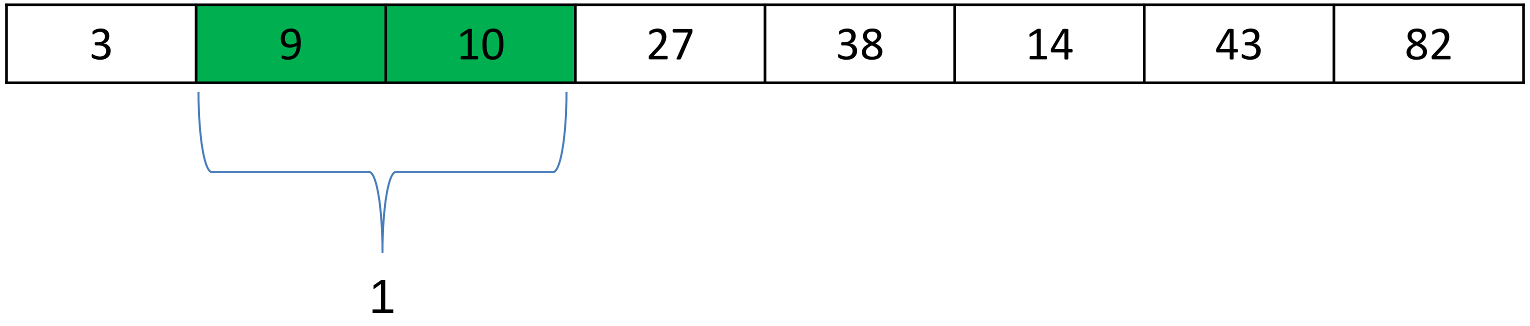
1

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.



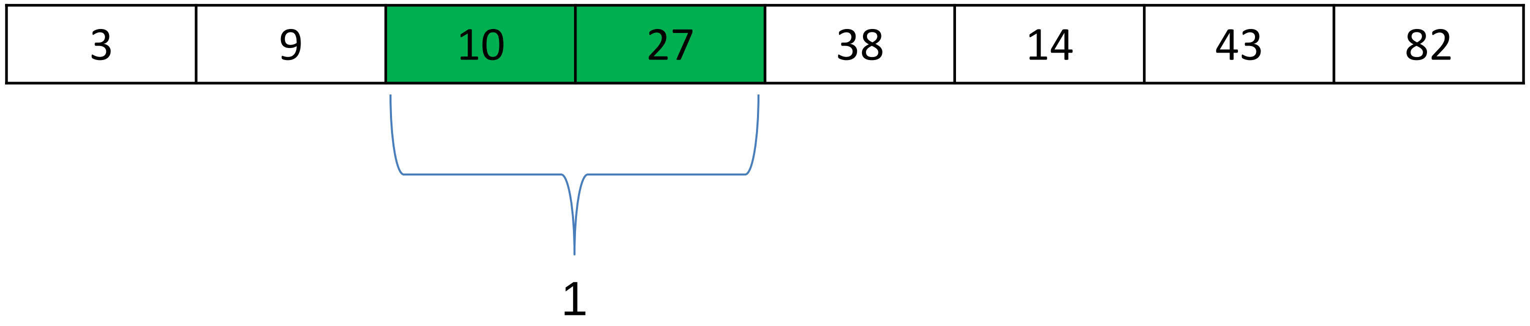
3	9	10	27	38	14	43	82

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.



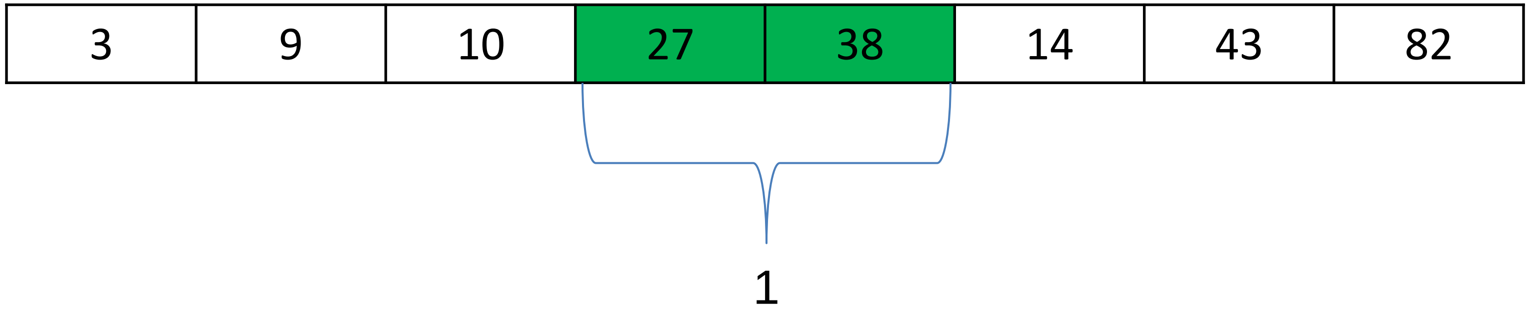
3	9	10	27	38	14	43	82

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.



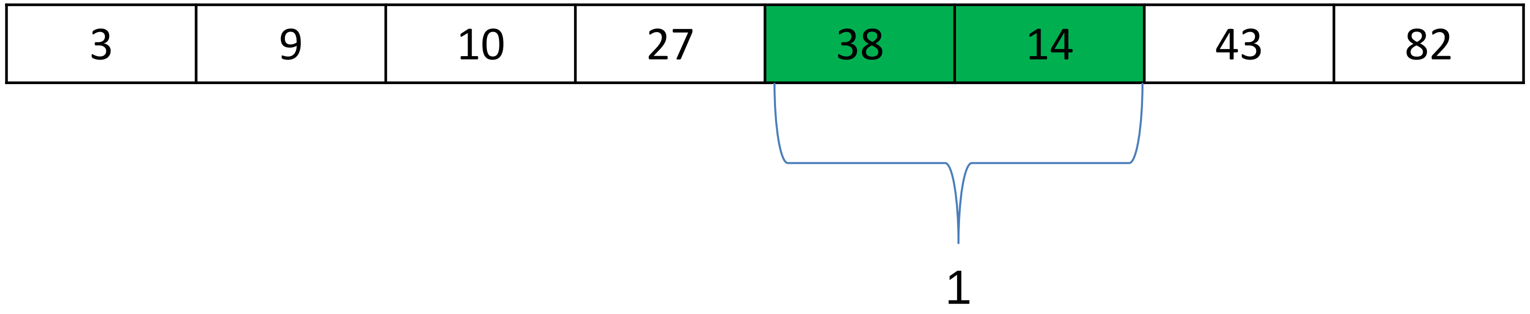
3	9	10	🟩 27	🟩 38	14	43	82

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.



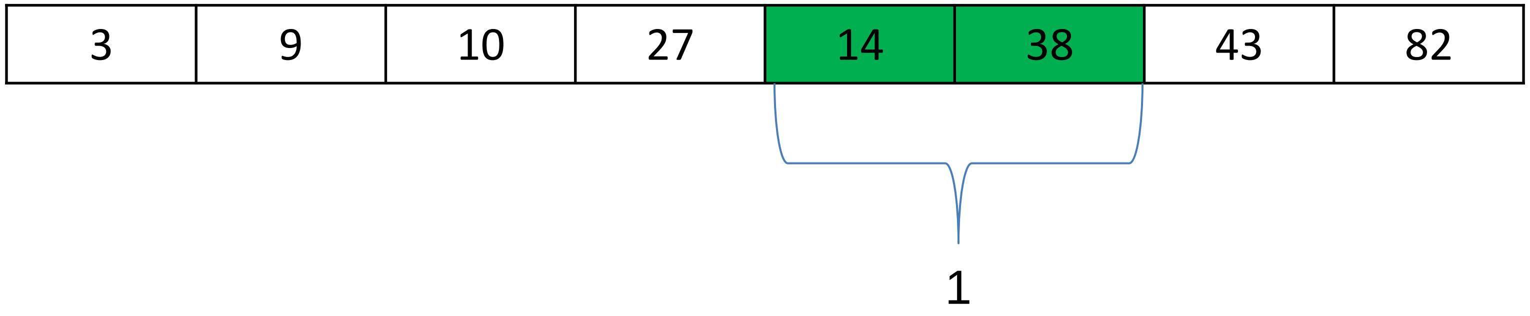
3	9	10	27	🟩 38	🟩 14	43	82

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.



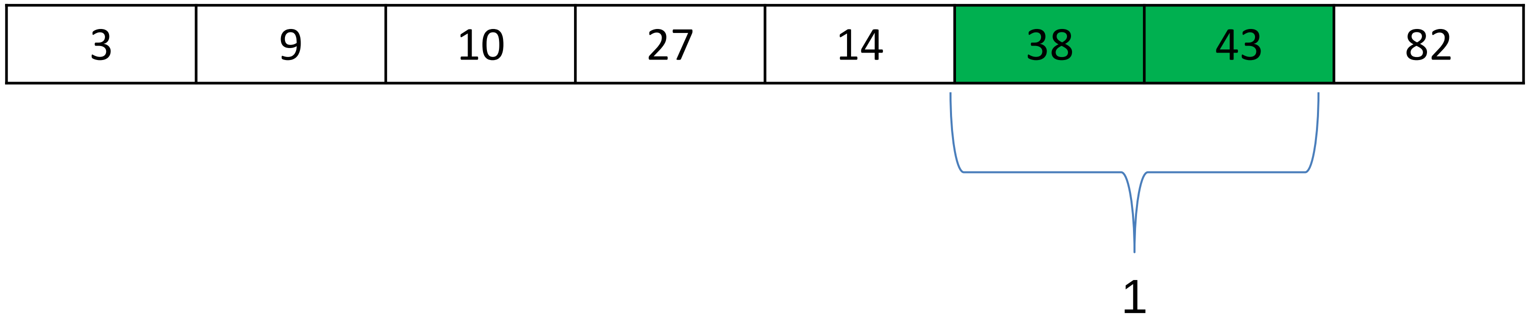
3	9	10	27	🟩 14	🟩 38	43	82

N = 8

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

Shell Sort

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



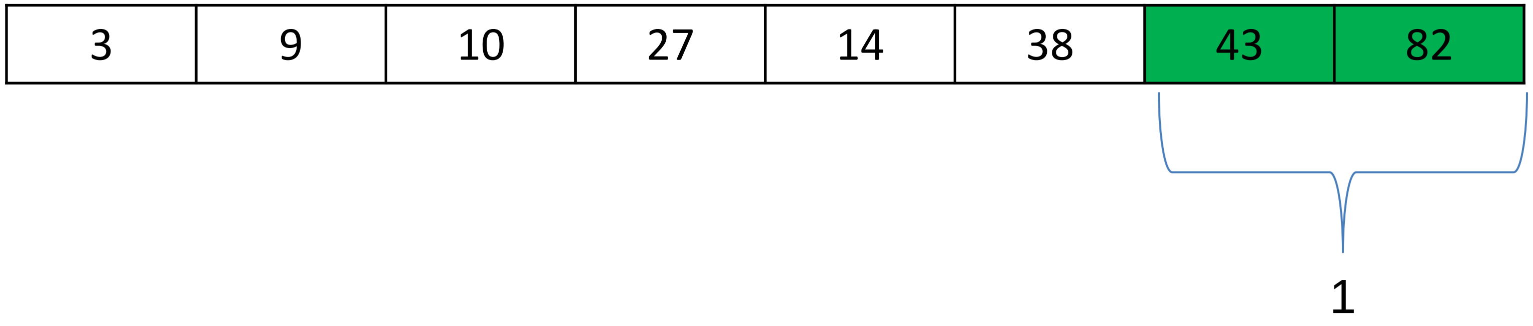
3	9	10	27	14	38	43	82

N = 8

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

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

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

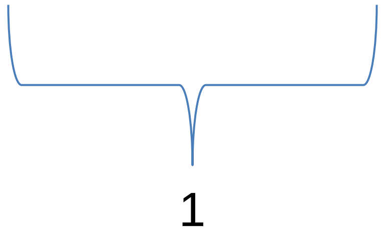
~~Gap = 4~~
~~Gap = 2~~
Gap = 1

(do it again ??)

Shell Sort

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

3	9	10	27	14	38	43	82



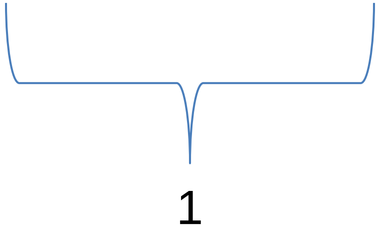
N = 8

~~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



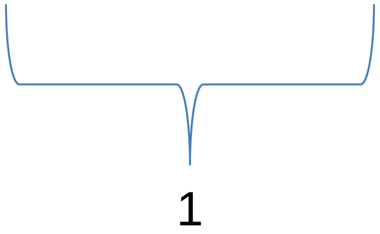
N = 8

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

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

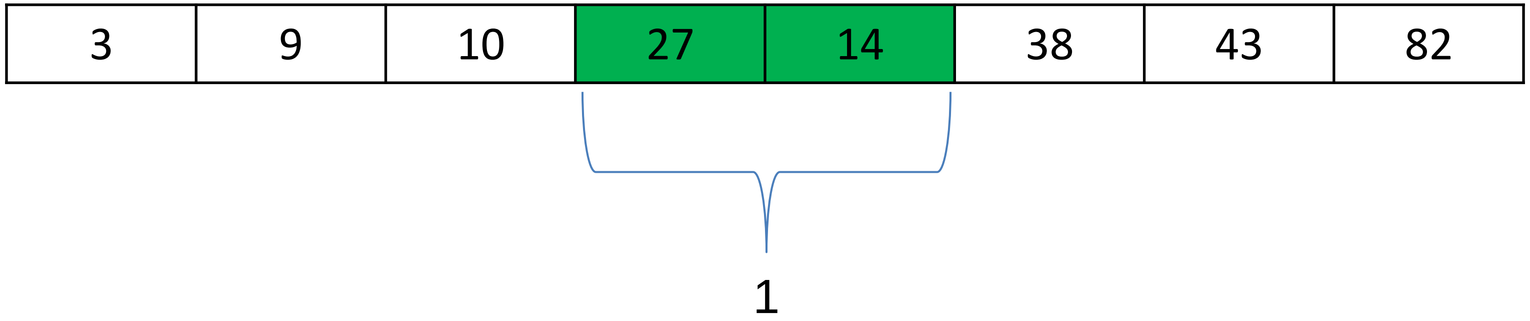


N = 8

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

Shell Sort

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



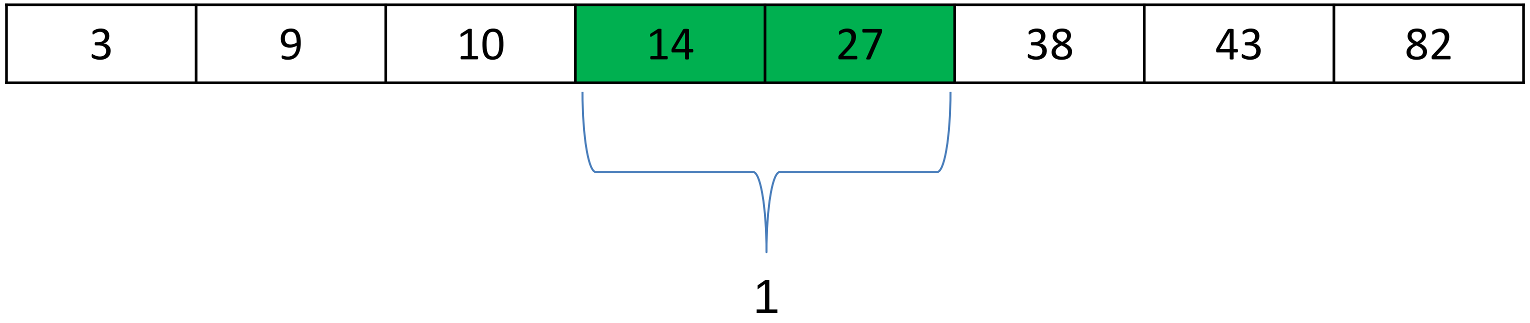
3	9	10	27	14	38	43	82

N = 8

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

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

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)

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) if we don’t keep track of permutations checked

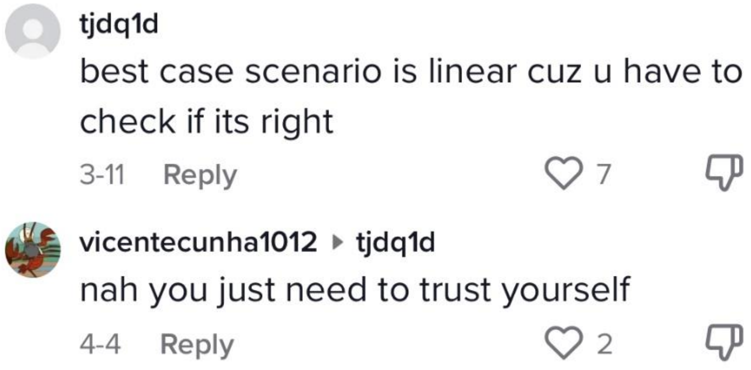
O(n!) if we keep track of permuations

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!



tjdq1d

best case scenario is linear cuz u have to check if its right

3-11 Reply ♡ 7

vicentecunha1012 ▸ tjdq1d

nah you just need to trust yourself

4-4 Reply ♡ 2

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