Shell Sort Algorithm

History and Concept

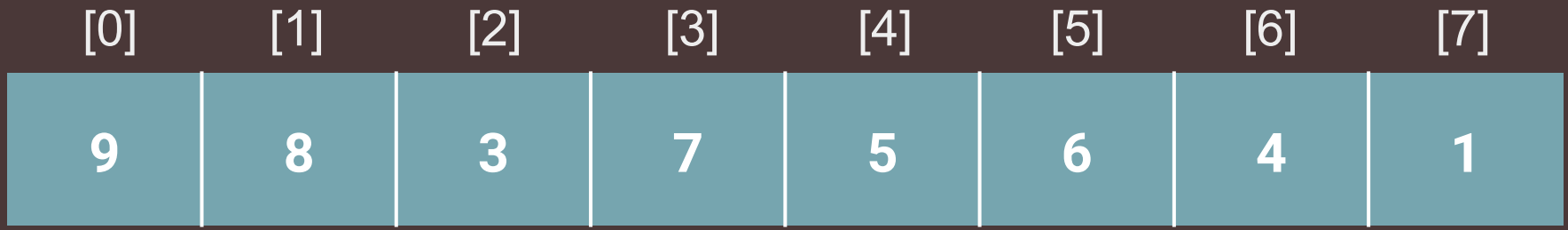
- Invented by Donald Lewis Shell
- In-place comparison sort
- It’s running time depends heavily on the gap sequence/interval
  - *More on this later on in our discussion*

Shell Sort In Essence

- Shell sort is an algorithm that first sorts the elements far apart from each other and successively reduces the interval between the elements to be sorted. It is a generalized version of insertion sort.
- In shell sort, elements at a specific interval are sorted. The interval between the elements is gradually decreased based on the sequence used.

  Interval - a space between two things; a gap.

How Insertion Sort Works



N = 8

Where N is the number of elements and the size of our array

How Insertion Sort Works

1. Select the first unsorted element.

2. Swap left elements to the right and insert the unsorted element into the correct position.

3. Advance the position by one element (i++).

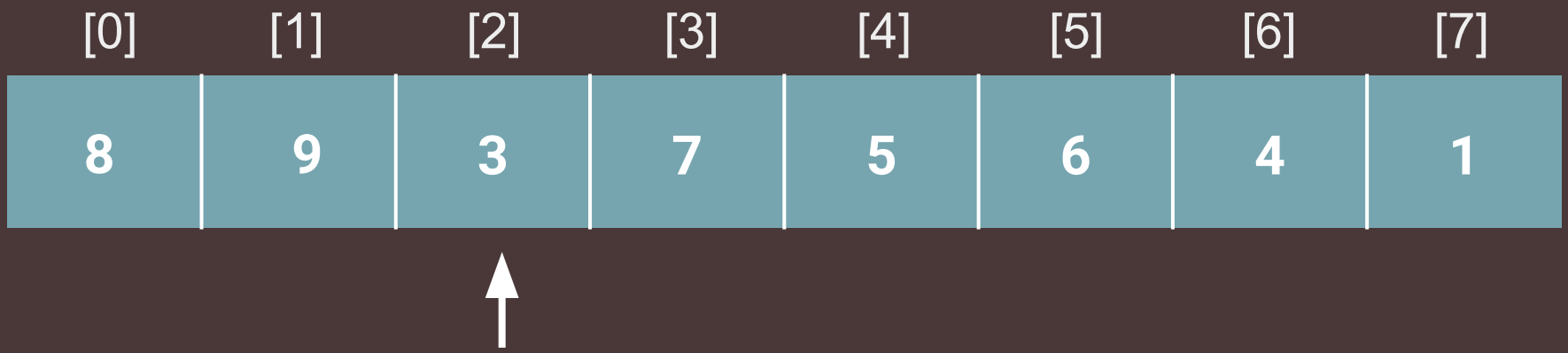
temp = **8**

How Insertion Sort Works

1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

temp = 8

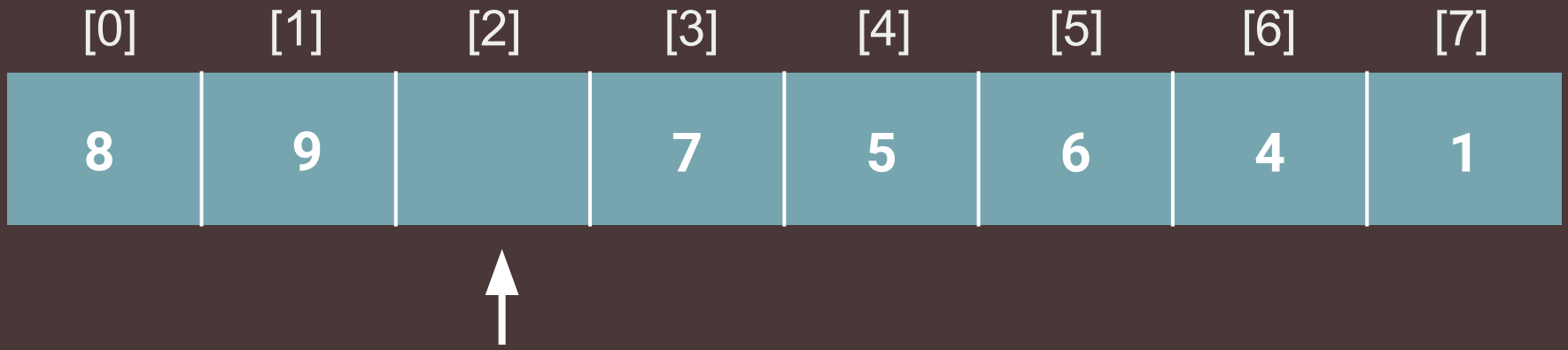
How Insertion Sort Works



1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. **Advance the position by one element (i++).**

temp = **8**

How Insertion Sort Works



1. Select the first unsorted element.

2. Swap left elements to the right and insert the unsorted element into the correct position.

3. Advance the position by one element (i++).

temp = 3

How Insertion Sort Works

1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

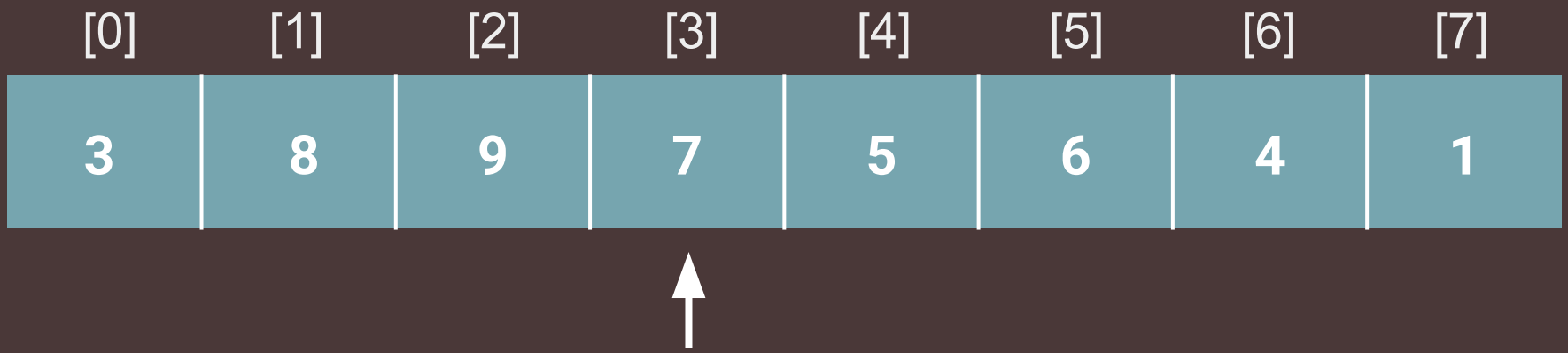
temp = 3

How Insertion Sort Works

1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

temp = 3

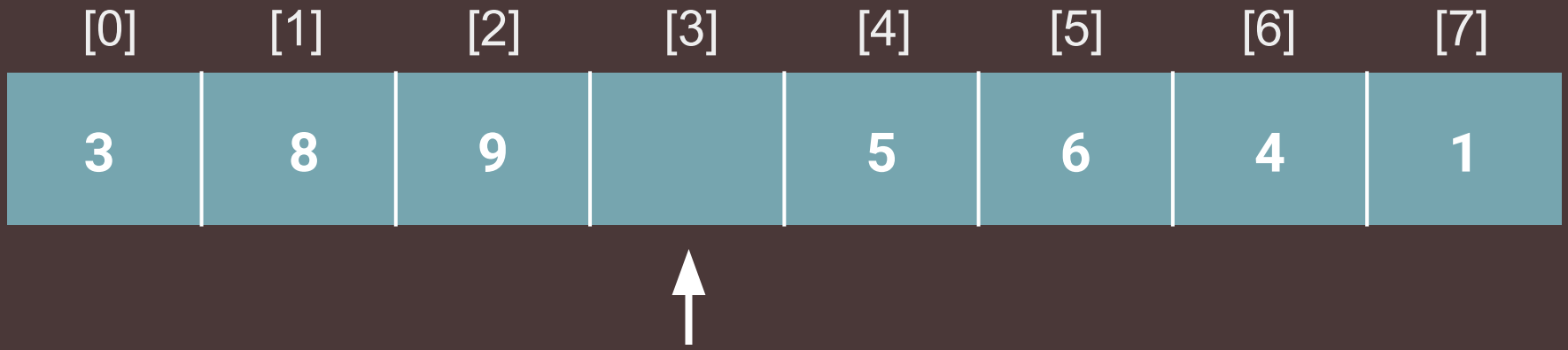
How Insertion Sort Works



1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. **Advance the position by one element (i++).**

temp = 3

How Insertion Sort Works



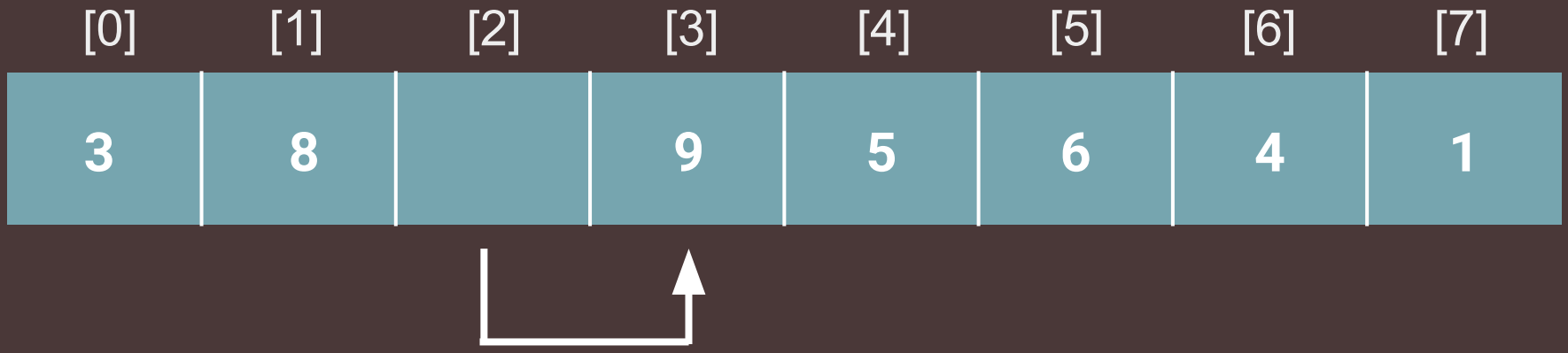
1. Select the first unsorted element.

2. Swap left elements to the right and insert the unsorted element into the correct position.

3. Advance the position by one element (i++).

temp = 7

How Insertion Sort Works



1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

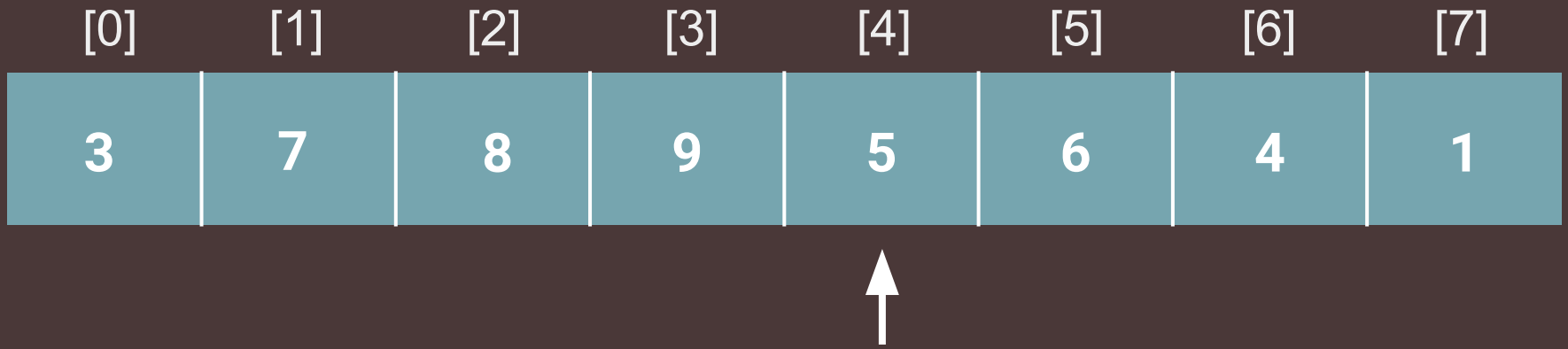
temp = 7

How Insertion Sort Works

1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

temp = 7

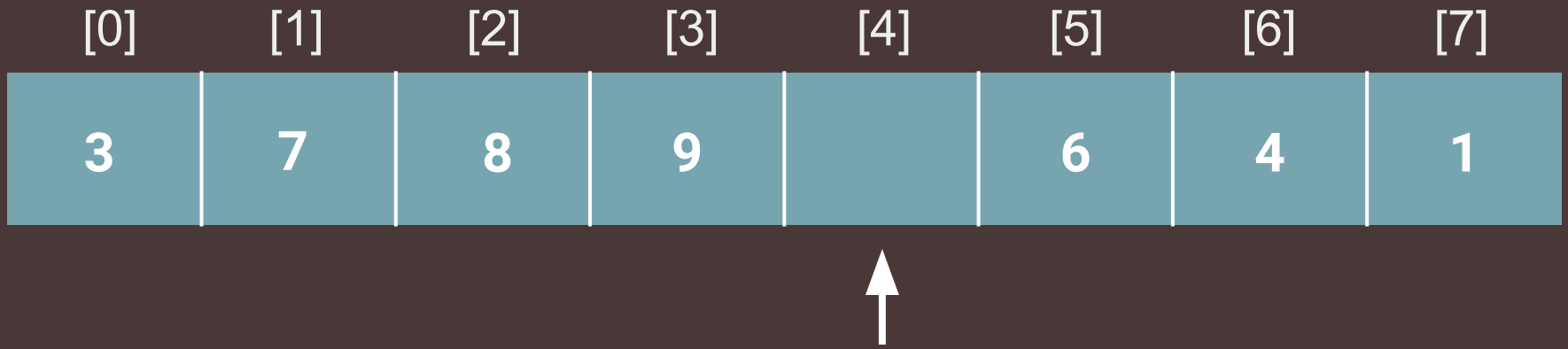
How Insertion Sort Works



1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. **Advance the position by one element (i++).**

temp = 7

How Insertion Sort Works



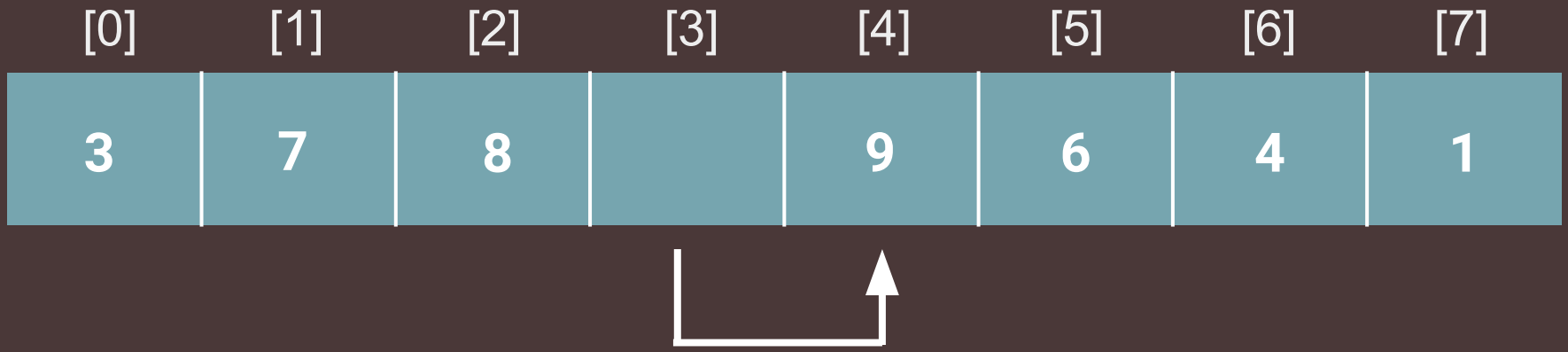
1. Select the first unsorted element.

2. Swap left elements to the right and insert the unsorted element into the correct position.

3. Advance the position by one element (i++).

temp = **5**

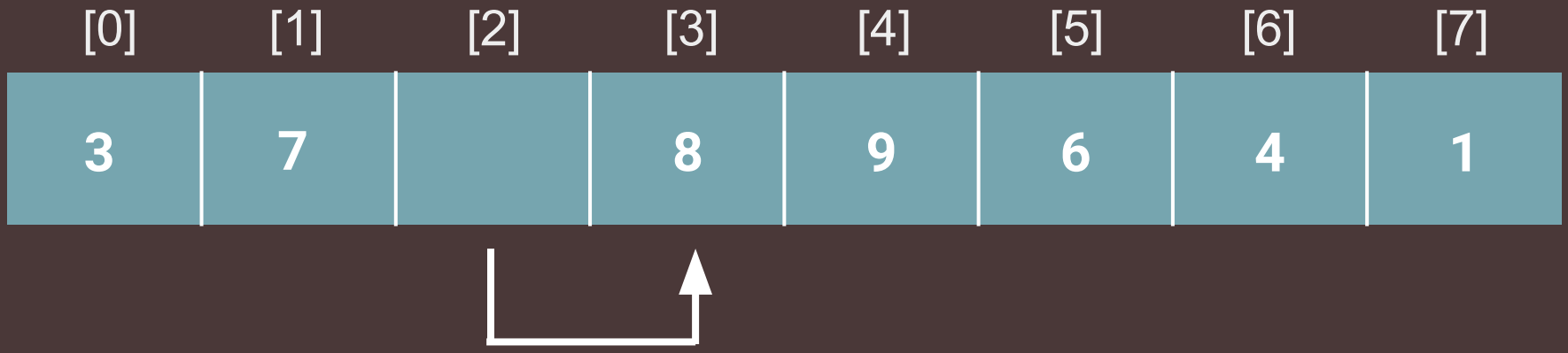
How Insertion Sort Works



1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

temp = **5**

How Insertion Sort Works



1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

temp = **5**

How Insertion Sort Works

1. Select the first unsorted element.
2. Swap left elements to the right and insert the unsorted element into the correct position.
3. Advance the position by one element (i++).

temp = **5**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
3	5	7	8	9	6	4	1

6th Iteration

3	5	6	7	8	9	4	1

7th Iteration

3	4	5	6	7	8	9	1

8th Iteration

1	3	4	5	6	7	8	9

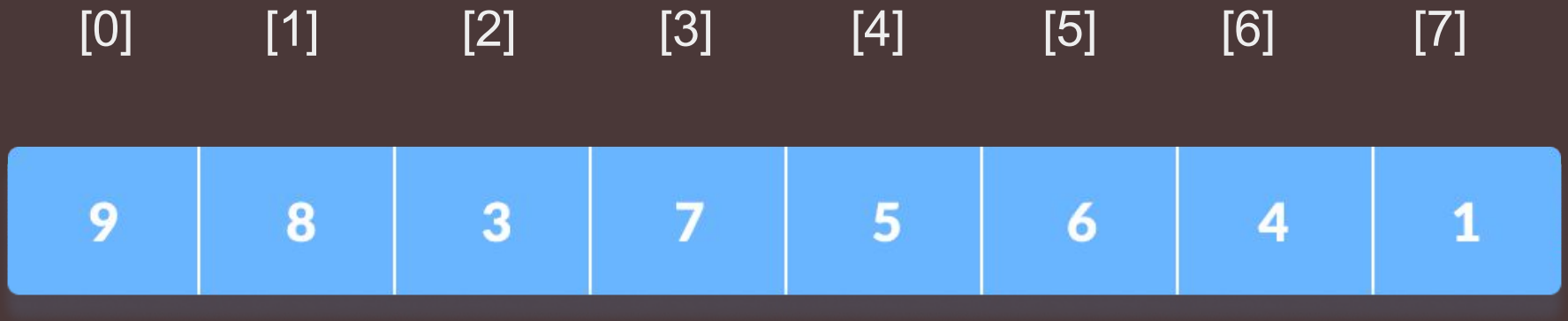
Insertion Sort In Code

Time Complexity: Avg Case: **O(n²)** Best Case: **O(n)**

```
void insertionSort(int arr[]){
    int i, j, temp;
    for(i = 1; i < SIZE; i++){
        temp = arr[i];
        for(j = i; j > 0 && arr[j - 1] > temp; j--){
            arr[j] = arr[j - 1];
        }
        arr[j] = temp;
    }
}
```

How Shell Sort Works

1. Suppose, we need to sort the following array.



Wherein N (is our num. of elements) so N = 8

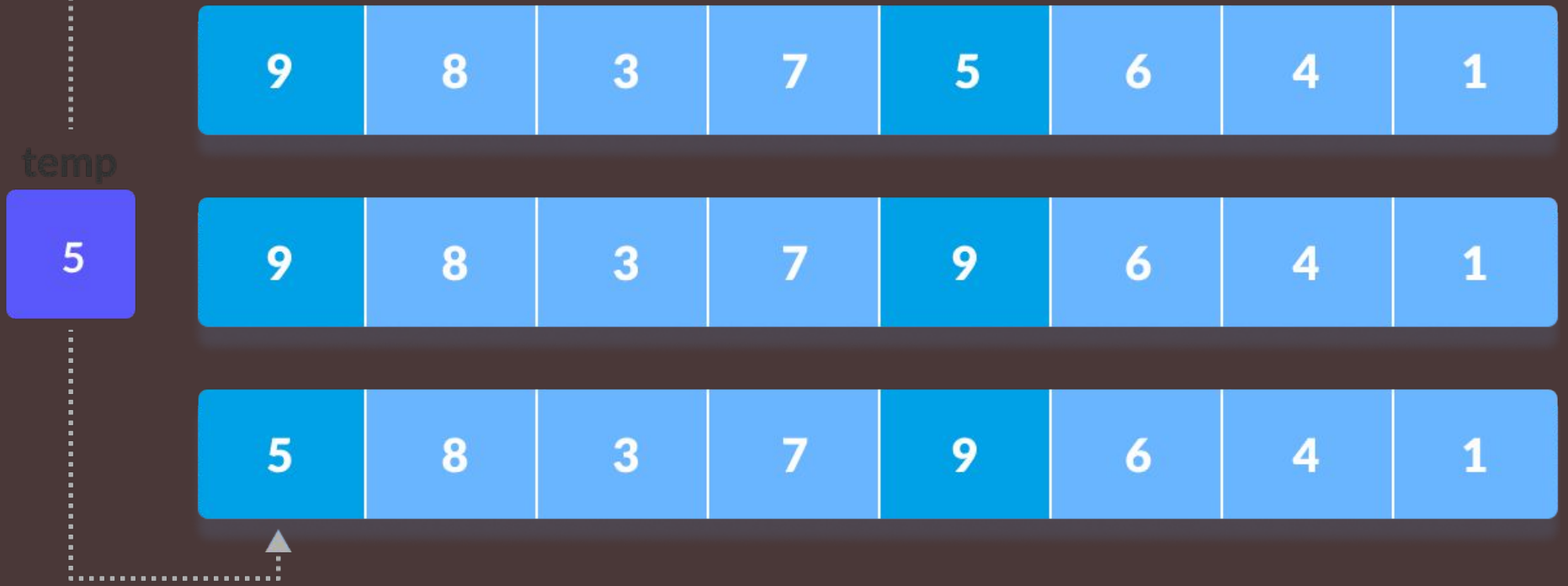
How Shell Sort Works

2. In the first loop, if the array size is N = 8 then, the elements lying at the interval of N/2 = 4 are compared and swapped if they are not in order.

a.The 0th element is compared with the 4th element.

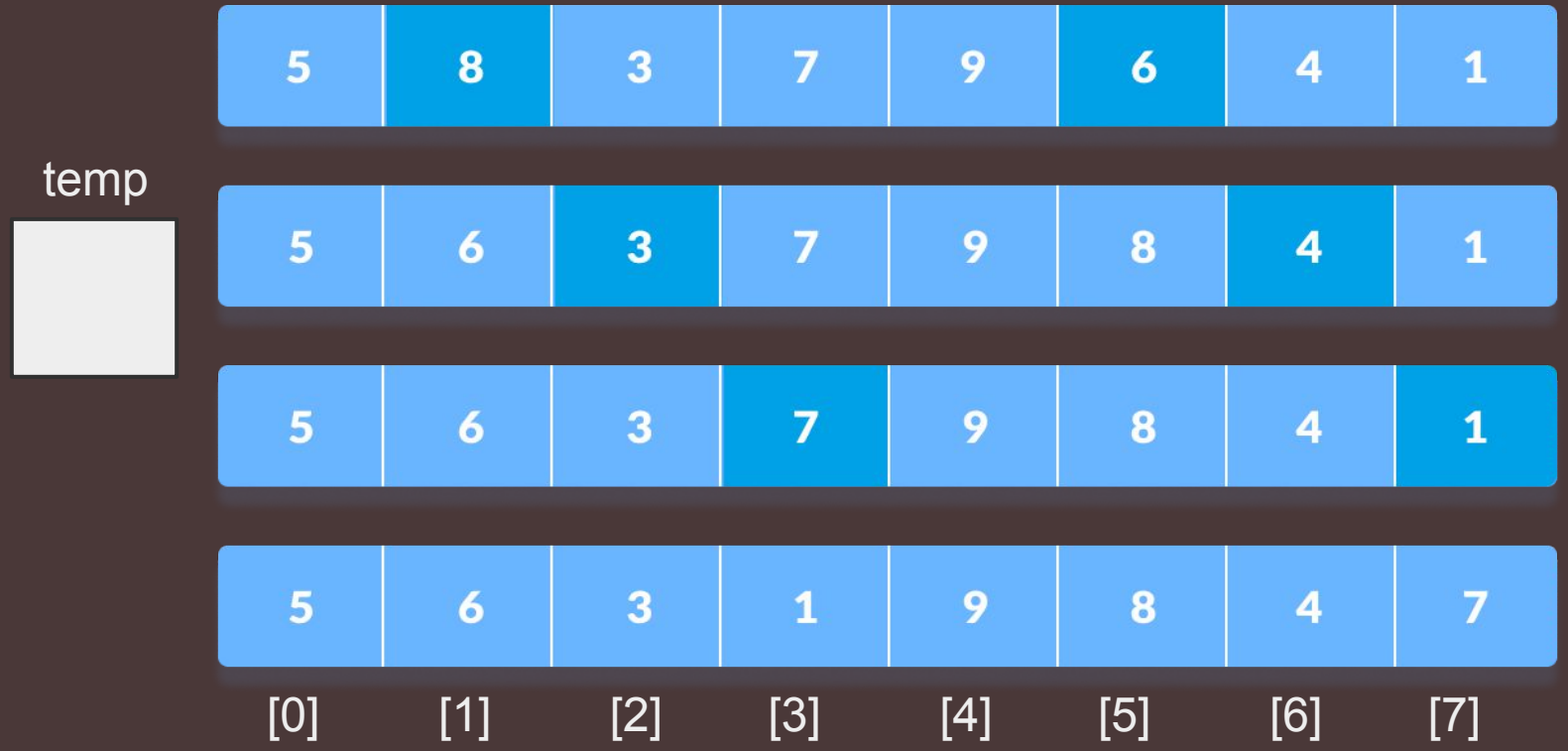
b. If the 0th element is greater than the 4th one then, the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position. (I want everyone to make a variable j, to keep track of what index we're currently at… right now j = 4)

How Shell Sort Works



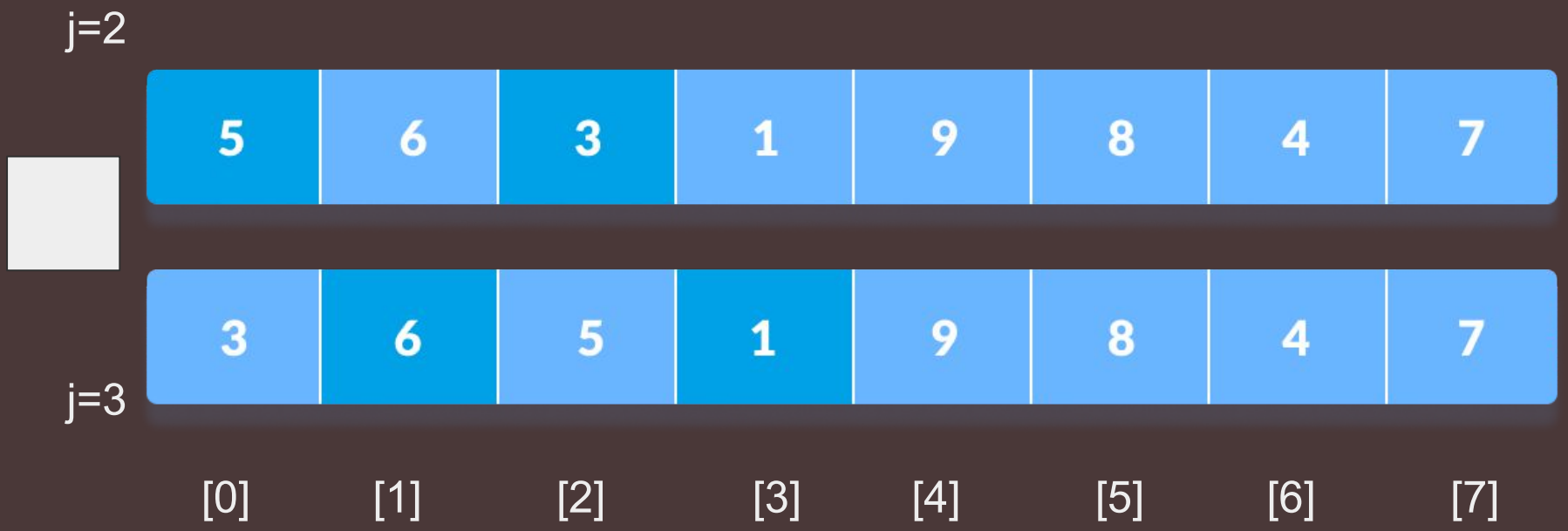
How Shell Sort Works

P.S. THIS PROCESS GOES ON FOR ALL REMAINING ELEMENTS



How Shell Sort Works

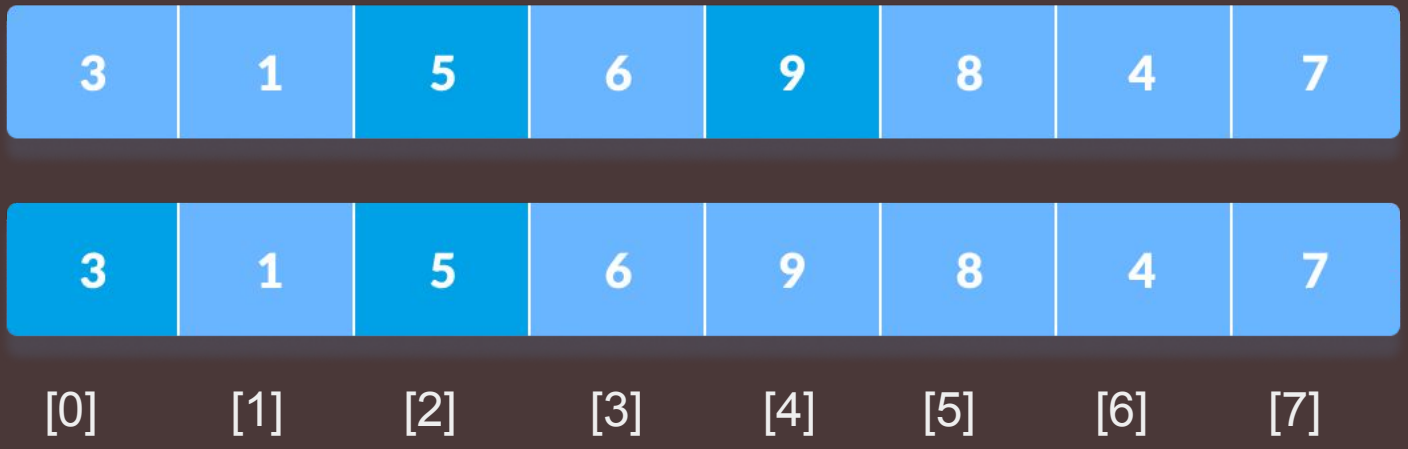
3. In the second iteration of this loop, an interval of N/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted.



How Shell Sort Works

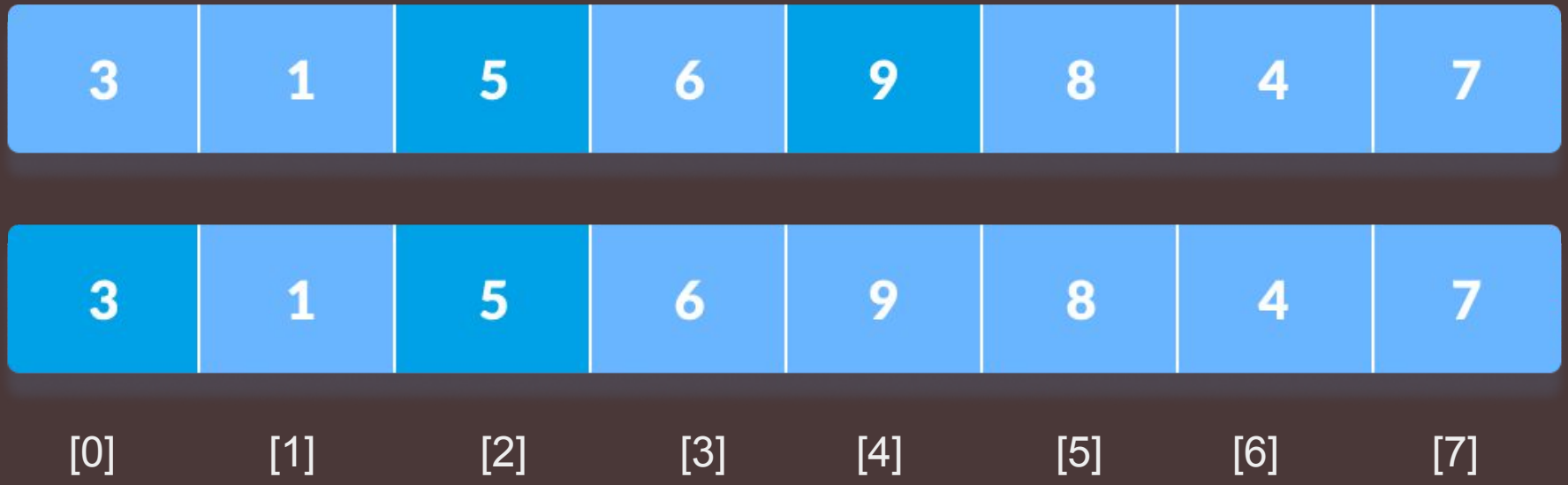
You might get confused at this point. It’s fine - life goes on.

j=4...2



All the elements in the array lying at the current interval are compared.

How Shell Sort Works



The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position are also compared. All the elements in the array lying at the current interval are compared.

— How Shell Sort Works

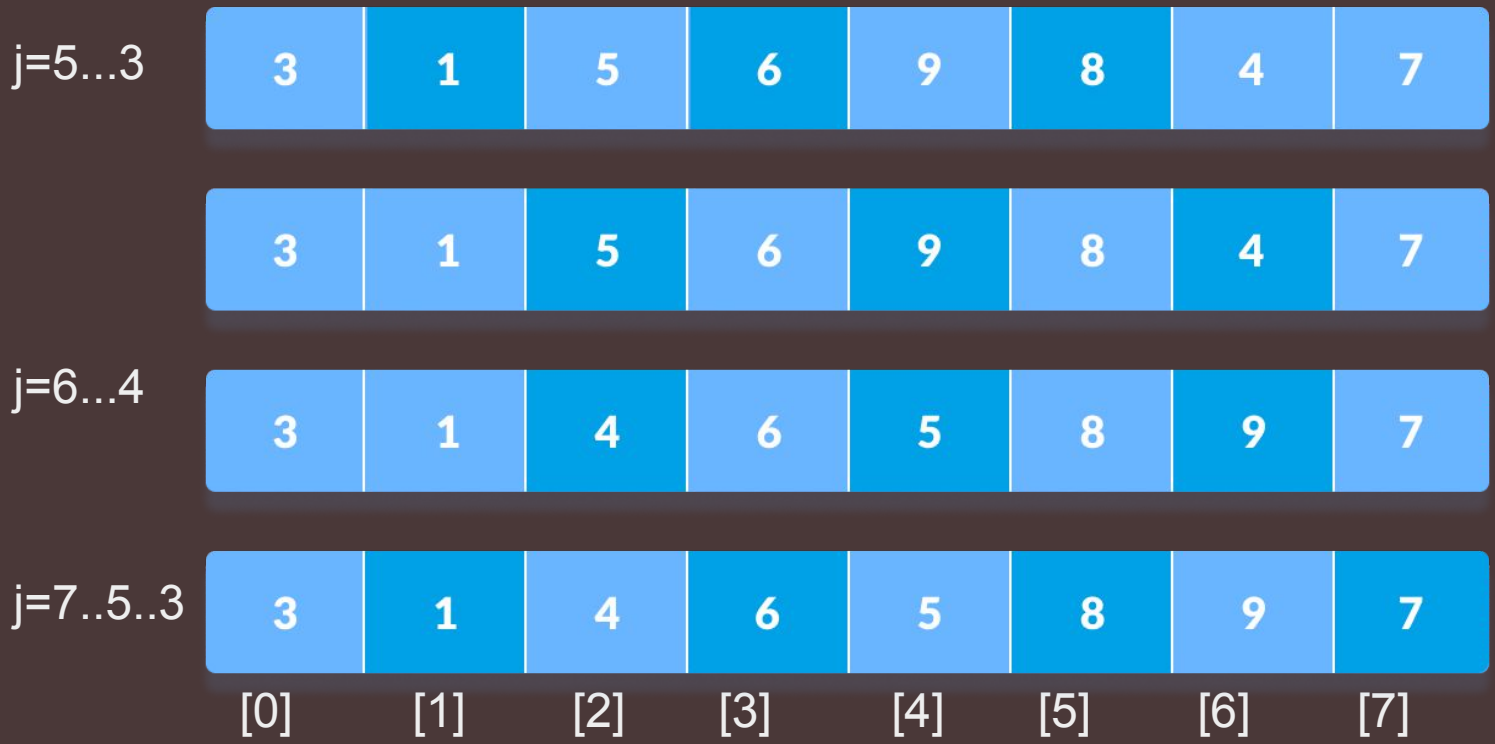
The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position are also compared. All the elements in the array lying at the current interval are compared…. **WHAT DO YOU MEAN?????**

- How can you tell if the index of an array is lying at the current interval.
  - Recall that we have a gap value, and a J variable which will hold whatever is the index of the element we want to be sorted with…
  - **If J-gap >= gap** is true , then **J-gap** is still within the range of the current interval
    - Earlier, **J = 4 …** so J-=gap which means J = 4 - 2 = 2…
    - Is J >= gap ? 2 >= 2 … so YES.
      - We then backtrack/go backwards…. Now comparing element of [2] and [0]

“IS THERE AN ELEMENT J-gap SPACES BEFORE MY CURRENT POSITION?”

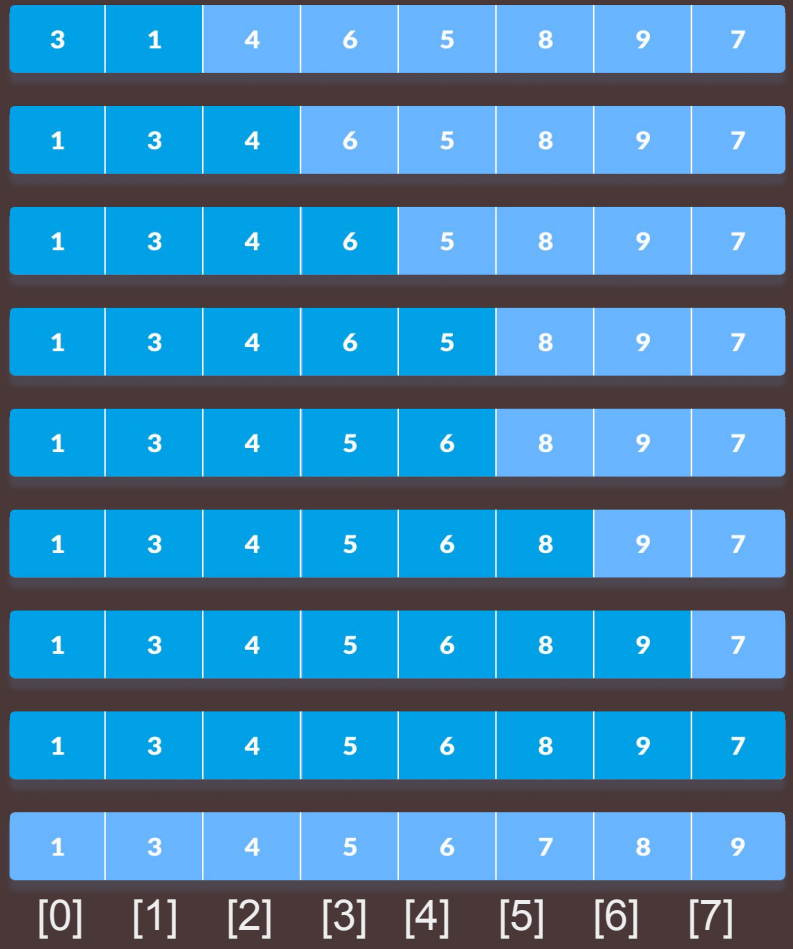
How Shell Sort Works

The same process goes for the remaining elements of that interval



How Shell Sort Works

4. FINALLY, when the interval is N/8 = 8/8 = 1, then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.


3 1 4 6 5 8 9 7
1 3 4 6 5 8 9 7
1 3 4 6 5 8 9 7
1 3 4 6 5 8 9 7
1 3 4 5 6 8 9 7
1 3 4 5 6 8 9 7
1 3 4 5 6 8 9 7
1 3 4 5 6 8 9 7
1 3 4 5 6 7 8 9
[0] [1] [2] [3] [4] [5] [6] [7]

Pseudocode of Shell Sort Algorithm

```
shellSort(array, size)
  for interval i <- size/2n down to 1
    for each interval "i" in array
        sort all the elements at interval "i"
  end shellSort
```

TLDR: *You will need 1 loop to keep track of interval, another to keep track of your interval ‘i’ and anothah one to backtrack/go backwards when in range of interval*

Let’s

SIMULATE ANOTHA ONE

Running Time

The best case ∈ O(nlogn):

The best-case is when the array is already sorted. This would mean that the 2nd condition of our 3rd loop will never be true, making the inner loop a constant time operation. Using the bounds you've used for the other loops gives O(nlogn). The best case of O(n) is reached by using a constant number of increments.

TDLR: *A situation wherein the array is already sorted, so only our 1st and second loop will keep iterating, never will we enter the statements in the 3rd loop*

Running Time

The worst-case ∈ O(n^2):

Given your upper bound for each loop you get O((log n)n^2) for the worst-case. But add another variable for the gap size g. The number of compare/exchanges needed in the inner while is now <= n/g. The number of compare/exchanges of the middle while is <= n^2/g. Add the upper-bound of the number of compare/exchanges for each gap together: n^2 + n^2/2 + n^2/4 + ... <= 2n^2 ∈ O(n^2). This matches the known worst-case complexity for the gaps you've used.

TDLR: *When the array's elements are all misplaced, meaning nothing is in the proper index yet. Get rekt.*

Running Time

The worst case ∈ Ω(n^2):

Consider the array where all the even positioned elements are greater than the median. The odd and even elements are not compared until we reach the last increment of 1. The number of compare/exchanges needed for the last iteration is Ω(n^2).

TDLR: *A scenario wherein the even-number index's elements are already in the appropriate positions or are always greater than the former index we compare it with.Basically the only time we actually sort is when interval/gap is 1. (insertion time)*

Sources

Weiss, M. (1993). Sorting. In *Data Structures and Algorithm Analysis in C* (pp. 222-223). Redwood City, California: The Benjamin/Cummings Publishing.

https://www.programiz.com/dsa/shell-sort#:~:text=Shell%20sort%20is%20an%20algorithm,a%20specific%20interval%20are%20sorted