//Assignment no 10

import java.util.Scanner;

/// Java implementation of ShellSort

class ShellSort

{

/\* An utility function to print array of size n\*/

static void printArray(int arr[])

{

int n = arr.length;

for (int i=0; i<n; ++i)

System.out.print(arr[i] + " ");

System.out.println();

}

/\* function to sort arr using shellSort \*/

int sort(int arr[])

{

int n = arr.length;

// Start with a big gap, then reduce the gap

for (int gap = n/2; gap > 0; gap /= 2)

{

// Do a gapped insertion sort for this gap size.

// The first gap elements a[0..gap-1] are already

// in gapped order keep adding one more element

// until the entire array is gap sorted

for (int i = gap; i < n; i += 1)

{

// add a[i] to the elements that have been gap

// sorted save a[i] in temp and make a hole at

// position i

int temp = arr[i];

// shift earlier gap-sorted elements up until

// the correct location for a[i] is found

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

arr[j] = arr[j - gap];

// put temp (the original a[i]) in its correct

// location

arr[j] = temp;

}

}

return 0;

}

// Driver method

public static void main(String args[])

{

int n;

Scanner sc=new Scanner(System.in);

System.out.print("Enter the number of elements you want to store: ");

//reading the number of elements from the that we want to enter

n=sc.nextInt();

//creates an array in the memory of length 10

int[] arr = new int[n];

System.out.println("Enter the elements of the array: ");

for(int i=0; i<n; i++)

{

//reading array elements from the user

arr[i]=sc.nextInt();

}

System.out.println("Enter Array before sorting");

//int arr[] = {12, 34, 54, 2, 3};

System.out.println("Array before sorting");

printArray(arr);

ShellSort ob = new ShellSort();

ob.sort(arr);

System.out.println("Array after sorting");

printArray(arr);

}

}

output:

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ javac ShellSort.java

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ java ShellSort

Enter the number of elements you want to store: 5

Enter the elements of the array:

4

7

2

8

5

Enter Array before sorting

Array before sorting

4 7 2 8 5

Array after sorting

2 4 5 7 8

[gescoe@gescoe-OptiPlex-3010](mailto:gescoe@gescoe-OptiPlex-3010):~/Desktop/SE-A-55$

### ****Shell Sort: Theory and Algorithm****

#### ****Introduction:****

Shell Sort is an **insertion sort** variant that improves its efficiency by allowing the exchange of far-apart elements. It was introduced by **Donald Shell** in 1959. The key idea behind Shell Sort is to allow the exchange of elements that are far apart, which allows for faster sorting compared to a standard insertion sort.

The main difference between Shell Sort and regular insertion sort is that instead of comparing adjacent elements, Shell Sort compares elements that are far apart (based on a gap sequence), and as the algorithm proceeds, the gap reduces until the array is sorted.

#### ****How Shell Sort Works:****

1. **Gap Sequence**:
   * Instead of comparing adjacent elements, Shell Sort compares elements that are **distant** from each other. This distance is controlled by a **gap sequence**. The gap starts with a large value, and as the algorithm progresses, it reduces (usually halving the gap each time).
2. **Gapped Insertion Sort**:
   * Shell Sort is based on the **insertion sort** technique but with a twist: rather than comparing adjacent elements, the elements are compared based on the gap.
   * For a given gap, we perform an insertion sort on the subarrays formed by separating the original array into subarrays of size gap. After each pass, the gap is reduced, and the elements are progressively moved closer to their correct positions.
3. **Sorting Process**:
   * The algorithm starts with a large gap, so elements that are far apart are compared. This helps move larger elements towards the right positions more quickly, unlike regular insertion sort which only moves elements by one position at a time.
   * With each iteration, the gap decreases, and the algorithm gradually shifts towards performing a standard insertion sort.

#### ****Gap Sequence:****

The choice of gap sequence has a major impact on the performance of Shell Sort. Common gap sequences are:

* **Original sequence (Shell's original gap sequence)**: The gap starts with n/2, and in each iteration, it is halved (n/4, n/8, ...).
* **Hibbard's sequence**: The gap sequence follows the formula 1, 3, 7, 15, 31, ..., which ensures better performance than the original sequence.
* **Sedgewick's sequence**: A more advanced sequence which provides better worst-case performance.

#### ****Algorithm:****

Here’s a high-level explanation of the algorithm:

1. **Input**: An array of n elements.
2. **Initialize the gap sequence**: Start with a gap of n/2, and reduce the gap after each iteration (usually halving the gap).
3. **Perform Gapped Insertion Sort**:
   * For each gap, sort the array in a way that elements that are gap apart are compared and inserted into their correct position (like a normal insertion sort).
4. **Repeat** until the gap becomes 1, at which point the array is fully sorted.

#### ****Pseudocode:****

Here is a simple pseudocode for Shell Sort:

function shellSort(arr):

n = length(arr)

// Start with a big gap, then reduce the gap

gap = n / 2

while gap > 0:

// Perform a gapped insertion sort for the current gap

for i = gap to n-1:

temp = arr[i]

j = i

// Shift elements that are gap apart

while j >= gap and arr[j - gap] > temp:

arr[j] = arr[j - gap]

j -= gap

arr[j] = temp

// Reduce the gap for the next iteration

gap = gap / 2

return arr

#### ****Time Complexity:****

* **Best Case**: O(nlog⁡n)O(n \log n) (when the gap sequence is optimal)
* **Worst Case**: O(n2)O(n^2) (for the original gap sequence)
* **Average Case**: O(n3/2)O(n^{3/2}) for the original gap sequence. Using more sophisticated gap sequences can improve this to O(nlog⁡2n)O(n \log^2 n).

#### ****Space Complexity:****

* **Space Complexity**: O(1)O(1), since Shell Sort is an **in-place sorting algorithm** and does not require additional memory beyond the input array.

#### ****Advantages of Shell Sort:****

1. **Faster than Insertion Sort**: By allowing elements to move over a larger range, Shell Sort is faster than insertion sort, especially for larger arrays.
2. **Simple to Implement**: It is relatively simple compared to more complex sorting algorithms like Quick Sort or Merge Sort.
3. **In-place Sorting**: It does not require additional memory, which makes it space-efficient.

#### ****Disadvantages of Shell Sort:****

1. **Unpredictable Performance**: The performance of Shell Sort depends heavily on the gap sequence. Using a poor gap sequence can result in inefficient sorting.
2. **Not Stable**: Shell Sort is not a stable sort, meaning that it might not preserve the relative order of equal elements in the array.

#### ****Conclusion:****

Shell Sort is a relatively simple algorithm that can be effective for medium-sized arrays. It improves upon the traditional insertion sort by breaking the array into subarrays with a gap and sorting them. The performance of the algorithm depends significantly on the gap sequence, and by choosing the right gap sequence, Shell Sort can be made much faster than regular insertion sort, but it still doesn't reach the efficiency of more advanced algorithms like Quick Sort or Merge Sort.