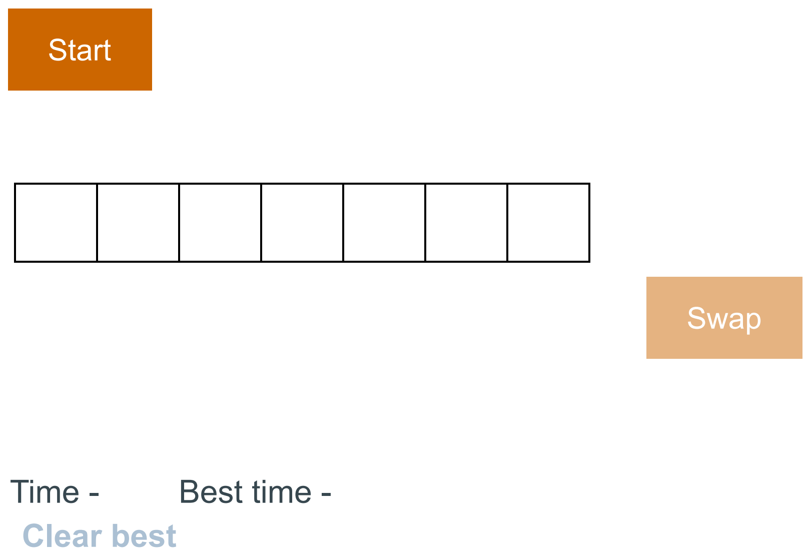
3.1 Sorting: Introduction

Sorting is the process of converting a list of elements into ascending (or descending) order. For example, given a list of numbers (17, 3, 44, 6, 9), the list after sorting is (3, 6, 9, 17, 44). You may have carried out sorting when arranging papers in alphabetical order, or arranging envelopes to have ascending zip codes (as required for bulk mailings).

The challenge of sorting is that a program can't "see" the entire list to know where to move an element. Instead, a program is limited to simpler steps, typically observing or swapping just two elements at a time. So sorting just by swapping values is an important part of sorting algorithms.

PARTICIPATION ACTIVITY | 3.1.1: Sort by swapping tool.

Sort the numbers from smallest on left to largest on right. Select two numbers then click "Swap values".



PARTICIPATION ACTIVITY | 3.1.2: Sorted elements.

1) The list is sorted into ascending order:
(3, 9, 44, 18, 76)

○ True

○ False

2) The list is sorted into descending order:
(20, 15, 10, 5, 0)

○ True

○ False

3) The list is sorted into descending order:
(99.87, 99.02, 67.93, 44.10)

○ True

○ False

4) The list is sorted into descending order:
(F, D, C, B, A)

○ True

○ False

5) The list is sorted into ascending order:
(chopsticks, forks, knives, spork)

○ True

○ False

6) The list is sorted into ascending order:
(great, greater, greatest)

○ True

○ False

3.2 Bubble sort

Bubble sort is a sorting algorithm that iterates through a list, comparing and swapping adjacent elements if the second element is less than the first element. Bubble sort uses nested loops. Given a list with N elements, the outer i-loop iterates N - 1 times. Each iteration moves the i^{th} largest element into sorted position. The inner j-loop iterates through all adjacent pairs, comparing and swapping adjacent elements as needed, except for the last i pairs that are already in the correct position.

Because of the nested loops, bubble sort has a runtime of O(N^2). Bubble sort is often considered impractical for real-world use because many faster sorting algorithms exist.

Figure 3.2.1: Bubble sort algorithm.

```
BubbleSort(numbers, numbersSize) {
   for (i = 0; i < numbersSize - 1; i++) {
      for (j = 0; j < numbersSize - i - 1; j++) {
         if (numbers[j] > numbers[j+1]) {
            temp = numbers[j]
            numbers[j] = numbers[j + 1]
            numbers[j + 1] = temp
         }
      }
   }
}
```

PARTICIPATION ACTIVITY 3.2.1: Bubble sort.

1) Bubble sort uses a single loop to sort the list.
   - True
   - False

2) Bubble sort only swaps adjacent elements.
   - True
   - False

3) Bubble sort's best and worst runtime complexity is O(N^2).
   - True
   - False

3.3 Selection sort

Selection sort

Selection sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly selects the proper next value to move from the unsorted part to the end of the sorted part.

PARTICIPATION ACTIVITY | 3.3.1: Selection sort.

Animation content:

undefined

Animation captions:

1. Selection sort treats the input as two parts, a sorted and unsorted part. Variables i and j keep track of the two parts.
2. The selection sort algorithm searches the unsorted part of the array for the smallest element; indexSmallest stores the index of the smallest element found.
3. Elements at i and indexSmallest are swapped.
4. Indices for the sorted and unsorted parts are updated.
5. The unsorted part is searched again, swapping the smallest element with the element at i.
6. The process repeats until all elements are sorted.

The index variable i denotes the dividing point. Elements to the left of i are sorted, and elements including and to the right of i are unsorted. All elements in the unsorted part are searched to find the index of the element with the smallest value. The variable indexSmallest stores the index of the smallest element in the unsorted part. Once the element with the smallest value is found, that element is swapped with the element at location i. Then, the index i is advanced one place to the right, and the process repeats.

The term "selection" comes from the fact that for each iteration of the outer loop, a value is selected for position i.

PARTICIPATION ACTIVITY | 3.3.2: Selection sort algorithm execution.

Assume selection sort's goal is to sort in ascending order.

1) Given list (9, 8, 7, 6, 5), what value will be in the 0th element after the first pass over the outer loop (i = 0)?

Check Show answer

2) Given list (9, 8, 7, 6, 5), how many

swaps will occur during the first pass of the outer loop (i = 0)?

Check Show answer

3) Given list (5, 9, 8, 7, 6) and i = 1, what will be the list after completing the second outer loop iteration? Type answer as: 1, 2, 3

Check Show answer

Selection sort runtime

Selection sort has the advantage of being easy to code, involving one loop nested within another loop, as shown below.

Figure 3.3.1: Selection sort algorithm.

```
SelectionSort(numbers, numbersSize) {
   i = 0
   j = 0
   indexSmallest = 0
   temp = 0  // Temporary variable for swap

   for (i = 0; i < numbersSize - 1; ++i) {

      // Find index of smallest remaining element
      indexSmallest = i
      for (j = i + 1; j < numbersSize; ++j) {

         if ( numbers[j] < numbers[indexSmallest] ) {
            indexSmallest = j
         }
      }

      // Swap numbers[i] and numbers[indexSmallest]
      temp = numbers[i]
      numbers[i] = numbers[indexSmallest]
      numbers[indexSmallest] = temp
   }
}

main() {
   numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 }
   NUMBERS_SIZE = 8
   i = 0

   print("UNSORTED: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   SelectionSort(numbers, NUMBERS_SIZE)

   print("SORTED: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()
}
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

Selection sort may require a large number of comparisons. The selection sort algorithm runtime is O(N^2). If a list has N elements, the outer loop executes N - 1 times. For each of those N - 1 outer loop executions, the inner loop executes an average of $\frac{N}{2}$ times. So the total number of comparisons is proportional to $(N-1) \cdot \frac{N}{2}$, or O(N^2). Other sorting algorithms involve more complex algorithms but have faster execution times.

PARTICIPATION ACTIVITY | 3.3.3: Selection sort runtime.

1) When sorting a list with 50 elements, `indexSmallest` will be assigned to a minimum of _____ times.

Check Show answer

2) How many times longer will sorting a list of 20 elements take compared to sorting a list of 10 elements?

Check Show answer

3) How many times longer will sorting a list of 500 elements take compared to a list of 50 elements?

Check Show answer

CHALLENGE ACTIVITY | 3.3.1: Selection sort.

361856.2187296.qx3zqy7

Start

When using selection sort to sort a list with 27 elements, what is the minimum number of assignments to `indexSmallest` once the outer loop starts?

Ex: 4

1 2 3 4

Check Next

3.4 Java: Selection sort

Selection sort

The selectionSort() method takes the array as a parameter, and has no return value since the array is sorted in-place by the algorithm.

The outer loop uses the variable i to hold the index position that will be sorted next in the array. The inner loop uses the variable j to examine all indices from i+1 to the end of the array. When the j loop finishes, the variable indexSmallest will store the index position of the smallest item in the array from i onward. The final step is to swap the values at position i and indexSmallest.

Figure 3.4.1: Selection sort algorithm.

```
import java.util.Arrays;

public class SelectionSortDemo {
   private static void selectionSort(int[] numbers) {
      for (int i = 0; i < numbers.length - 1; i++) {
         // Find index of smallest remaining element
         int indexSmallest = i;
         for (int j = i + 1; j < numbers.length; j++) {
            if (numbers[j] < numbers[indexSmallest]) {
               indexSmallest = j;
            }
         }

         // Swap numbers[i] and numbers[indexSmallest]
         int temp = numbers[i];
         numbers[i] = numbers[indexSmallest];
         numbers[indexSmallest] = temp;
      }
   }

   public static void main(String[] args) {
      // Create an array of numbers to sort
      int[] numbers = { 10, 2, 78, 4, 45, 32, 7, 11 };

      // Display the contents of the array
      System.out.println("UNSORTED: " + Arrays.toString(numbers));

      // Call the selectionSort method
      selectionSort(numbers);

      // Display the sorted contents of the array
      System.out.println("SORTED: " + Arrays.toString(numbers));
   }
}
```

```
UNSORTED: [10, 2, 78, 4, 45, 32, 7, 11]
SORTED: [2, 4, 7, 10, 11, 32, 45, 78]
```

PARTICIPATION ACTIVITY 3.4.1: Selection sort algorithm.

Suppose the selectionSort() method is executed with the array (3, 12, 7, 2, 9, 14, 8).

1) When i is assigned with 0 in the outer loop, what is j assigned with in the inner loop?

- 0
- 1
- 3

2) What is the final value for the variable i

inside the inner loop?

- 5
- 6
- 7

3) Which statement best describes the following code fragment taken from the selectionSort() method?

```
int temp = numbers[i];
numbers[i] =
numbers[indexSmallest];
numbers[indexSmallest] = temp;
```

- Shifts the value at position indexSmallest to the right one space in the array.
- Tests to see if the value at numbers[indexSmallest] is smaller than the value at numbers[i].
- Swaps the values located at indices i and indexSmallest.

zyDE 3.4.1: Selection sort algorithm.

This program uses the selection sort algorithm to sort an array. The code has been mod
to also calculate how many item comparisons are done. You can try running the progra
different arrays to see how the total number of comparisons changes (or doesn't chang

SelectionSortDemo.java Load default templ

```
import java.util.Arrays;

public class SelectionSortDemo {
   private static int selectionSort(int[] numbers) {
      // A variable to hold the number of item comparisons
      int comparisons = 0;

      for (int i = 0; i < numbers.length - 1; i++) {
         // Find index of smallest remaining element
         int indexSmallest = i;
         for (int j = i + 1; j < numbers.length; j++) {
            comparisons++;
            if (numbers[j] < numbers[indexSmallest]) {
               indexSmallest = j;
            }
         }
```

Run

3.5 Insertion sort

Insertion sort algorithm

Insertion sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part.

PARTICIPATION ACTIVITY 3.5.1: Insertion sort.

Animation content:

`undefined`

Animation captions:

1. Variable i is the index of the first unsorted element. Since the element at index 0 is already sorted, i starts at 1.
2. Variable j keeps track of the index of the current element being inserted into the sorted part. If the current element is less than the element to the left, the values are swapped.
3. Once the current element is inserted in the correct location in the sorted part, i is incremented to the next element in the unsorted part.
4. If the current element being inserted is smaller than all elements in the sorted part, that element will be repeatedly swapped with each sorted element until index 0 is reached.
5. Once all elements in the unsorted part are inserted in the sorted part, the list is sorted.

The index variable i denotes the starting position of the current element in the unsorted part. Initially, the first element (i.e., element at index 0) is assumed to be sorted, so the outer for loop initializes i to 1. The inner while loop inserts the current element into the sorted part by repeatedly swapping the current element with the elements in the sorted part that are larger. Once a smaller or equal element is found in the sorted part, the current element has been inserted in the correct location and the while loop terminates.

Figure 3.5.1: Insertion sort algorithm.

```
InsertionSort(numbers, numbersSize) {
   i = 0
   j = 0
   temp = 0  // Temporary variable for swap

   for (i = 1; i < numbersSize; ++i) {
      j = i
      // Insert numbers[i] into sorted part
      // stopping once numbers[i] in correct position
      while (j > 0 && numbers[j] < numbers[j - 1]) {

         // Swap numbers[j] and numbers[j - 1]
         temp = numbers[j]
         numbers[j] = numbers[j - 1]
         numbers[j - 1] = temp
         --j
      }
   }
}

main() {
   numbers = { 10, 2, 78, 4, 45, 32, 7, 11 }
   NUMBERS_SIZE = 8
   i = 0

   print("UNSORTED: ")
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   InsertionSort(numbers, NUMBERS_SIZE)

   print("SORTED: ")
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()
}
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

PARTICIPATION ACTIVITY 3.5.2: Insertion sort algorithm execution.

Assume insertion sort's goal is to sort in ascending order.

1) Given list (20, 14, 85, 3, 9), what

value will be in the 0^{th} element after the first pass over the outer loop (i = 1)?

Check Show answer

2) Given list (10, 20, 6, 14, 7), what will be the list after completing the second outer loop iteration (i = 2)? Type answer as: 1, 2, 3

Check Show answer

3) Given list (1, 9, 17, 18, 2), how many swaps will occur during the outer loop execution (i = 4)?

Check Show answer

Insertion sort runtime

Insertion sort's typical runtime is O(N^2). If a list has N elements, the outer loop executes N - 1 times. For each outer loop execution, the inner loop may need to examine all elements in the sorted part. Thus, the inner loop executes on average $\frac{N}{2}$ times. So the total number of comparisons is proportional to $(N-1) \cdot (\frac{N}{2})$, or O(N^2). Other sorting algorithms involve more complex algorithms but faster execution.

PARTICIPATION ACTIVITY 3.5.3: Insertion sort runtime.

1) In the worst case, assuming each comparison takes 1 µs, how long will insertion sort algorithm take to sort a list of 10 elements?

µs

Check Show answer

2) Using the Big O runtime complexity, how many times longer will sorting a list of 20 elements take compared to sorting a list of 10 elements?

Check Show answer

Nearly sorted lists

For sorted or nearly sorted inputs, insertion sort's runtime is O(N). A ***nearly sorted*** list only contains a few elements not in sorted order. Ex: (4, 5, 17, 25, 89, 14) is nearly sorted having only one element not in sorted position.

PARTICIPATION ACTIVITY 3.5.4: Nearly sorted lists.

Determine if each of the following lists is unsorted, sorted, or nearly sorted. Assume ascending order.

1) (6, 14, 85, 102, 102, 151)
- Unsorted
- Sorted
- Nearly sorted

2) (23, 24, 36, 48, 19, 50, 101)
- Unsorted
- Sorted
- Nearly sorted

3) (15, 19, 21, 24, 2, 3, 6, 11)
- Unsorted
- Sorted
- Nearly sorted

Insertion sort runtime for nearly sorted input

For each outer loop execution, if the element is already in sorted position, only a single comparison is made. Each element not in sorted position requires at most N comparisons. If there are a constant number, C, of unsorted elements, sorting the N - C sorted elements requires one comparison each, and sorting the C unsorted elements requires at most N comparisons each. The runtime for nearly sorted inputs is O((N - C) * 1 + C * N) = O(N).

PARTICIPATION ACTIVITY 3.5.5: Using insertion sort for nearly sorted list.

Animation captions:

1. Sorted part initially contains the first element.
2. An element already in sorted position only requires a single comparison, which is O(1) complexity.
3. An element not in sorted position requires O(N) comparisons. For nearly sorted inputs, insertion sort's runtime is O(N).

PARTICIPATION ACTIVITY 3.5.6: Insertion sort algorithm execution for nearly sorted input.

Assume insertion sort's goal is to sort in ascending order.

1) Given list (10, 11, 12, 13, 14, 15), how many comparisons will be made during the third outer loop execution (i = 3)?

Check Show answer

2) Given list (10, 11, 12, 13, 14, 7), how many comparisons will be made during the final outer loop execution (i = 5)?

Check Show answer

3) Given list (18, 23, 34, 75, 3), how many total comparisons will insertion sort require?

Check Show answer

CHALLENGE ACTIVITY | 3.5.1: Insertion sort.

361856.2187296.qx3zqy7

Start

Given list (22, 26, 31, 36, 37, 23, 32, 35), what is the value of i when the first swap executes?

Ex: 1

1 2 3 4 5

Check Next

3.6 Java: Insertion sort

Insertion sort algorithm

The insertionSort() method has one parameter, `numbers`, which is an unsorted array of elements. The array is sorted in place.

The index variable `i` denotes the starting position of the current element in the unsorted part. Initially, the first element (i.e., element at index 0) is assumed to be sorted, so the outer for loop assigns `i` with 1 to begin. The inner while loop inserts the current element into the sorted part by repeatedly swapping the current element with the elements in the array's sorted part that are larger. Once a smaller or equal element is found in the array's sorted part, the current element has been inserted in the correct location and the while loop terminates.

Figure 3.6.1: Insertion sort algorithm.

```
import java.util.Arrays;

public class InsertionSortDemo {
   private static void insertionSort(int[] numbers) {
      for (int i = 1; i < numbers.length; i++) {
         int j = i;
         while (j > 0 && numbers[j] < numbers[j - 1]) {
            // Swap numbers[j] and numbers [j - 1]
            int temp = numbers[j];
            numbers[j] = numbers[j - 1];
            numbers[j - 1] = temp;
            j--;
         }
      }
   }

   public static void main(String[] args) {
      // Create an array of numbers to sort
      int[] numbers = { 10, 2, 78, 4, 45, 32, 7, 11 };

      // Display the contents of the array
      System.out.println("UNSORTED: " + Arrays.toString(numbers));

      // Call the insertionSort method
      insertionSort(numbers);

      // Display the sorted contents of the array
      System.out.println("SORTED: " + Arrays.toString(numbers));
   }
}
```

```
UNSORTED: [10 2, 78, 4, 45, 32, 7, 11]
SORTED: [2, 4, 7, 10, 11, 32, 45, 78]
```

PARTICIPATION ACTIVITY 3.6.1: Insertion sort algorithm implementation.

1) Which index variable denotes the starting position of the current element in the unsorted part?

- ○ i
- ○ j
- ○ k

2) Which variable is passed as an argument to the insertionSort() method?

- temp
- numbersLength
- numbers

3) If the numbers array has 4 elements in descending order, then the insertionSort() method does _____ swaps.

- 4
- 6
- 16

zyDE 3.6.1: Insertion sort algorithm.

InsertionSortDemo.java Load default templ

```java
import java.util.Arrays;

public class InsertionSortDemo {
   private static void insertionSort(int[] numbers) {
      for (int i = 1; i < numbers.length; i++) {
         int j = i;
         while (j > 0 && numbers[j] < numbers[j - 1]) {
            // Swap numbers[j] and numbers [j - 1]
            int temp = numbers[j];
            numbers[j] = numbers[j - 1];
            numbers[j - 1] = temp;
            j--;
         }
      }
   }

   public static void main(String[] args) {
```

Run

3.7 Shell sort

Shell sort's interleaved lists

Shell sort is a sorting algorithm that treats the input as a collection of interleaved lists, and sorts each list individually with a variant of the insertion sort algorithm. Shell sort uses gap values to determine the number of interleaved lists. A ***gap value*** is a positive integer representing the distance between elements in an interleaved list. For each interleaved list, if an element is at index i, the next element is at index i + gap value.

Shell sort begins by choosing a gap value K and sorting K interleaved lists in place. Shell sort finishes by performing a standard insertion sort on the entire array. Because the interleaved parts have already been sorted, smaller elements will be close to the array's beginning and larger elements towards the end. Insertion sort can then quickly sort the nearly-sorted array.

Any positive integer gap value can be chosen. In the case that the array size is not evenly divisible by the gap value, some interleaved lists will have fewer items than others.

PARTICIPATION ACTIVITY 3.7.1: Sorting interleaved lists with shell sort speeds up insertion sort.

Animation captions:

1. If a gap value of 3 is chosen, shell sort views the list as 3 interleaved lists. 56, 12, and 75 make up the first list, 42, 77, and 91 the second, and 93, 82, and 36 the third.
2. Shell sort will sort each of the 3 lists with insertion sort.
3. The result is not a sorted list, but is closer to sorted than the original. Ex: The 3 smallest elements, 12, 42, and 36, are the first 3 elements in the list.
4. Sorting the original array with insertion sort requires 17 swaps.
5. Sorting the interleaved lists required 4 swaps. Running insertion sort on the array requires 7 swaps total, far fewer than insertion sort on the original array.

PARTICIPATION ACTIVITY 3.7.2: Shell sort's interleaved lists.

For each question, assume a list with 6 elements.

1) With a gap value of 3, how many interleaved lists will be sorted?

- ○ 1
- ○ 2
- ○

○ 6

2) With a gap value of 3, how many items will be in each interleaved list?

○ 1

○ 2

○ 3

○ 6

3) If a gap value of 2 is chosen, how many interleaved lists will be sorted?

○ 1

○ 2

○ 3

○ 6

4) If a gap value of 4 is chosen, how many interleaved lists will be sorted?

○ A gap value of 4 cannot be used on an array with 6 elements.

○ 2

○ 3

○ 4

Insertion sort for interleaved lists

If a gap value of K is chosen, creating K entirely new lists would be computationally expensive. Instead of creating new lists, shell sort sorts interleaved lists in-place with a variation of the insertion sort algorithm. The insertion sort algorithm variant redefines the concept of "next" and "previous" items. For an item at index X, the next item is at X + K, instead of X + 1, and the previous item is at X - K instead of X - 1.

PARTICIPATION ACTIVITY 3.7.3: Interleaved insertion sort.

Animation content:

```
undefined
```

Animation captions:

1. Calling InsertionSortInterleaved with a start index of 0 and a gap of 3 sorts the interleaved list with elements at indices 0, 3, and 6. i and j are first assigned with index 3.
2. When swapping the 2 elements 45 and 88, 45 jumps the gap and moves towards the front more quickly compared to the regular insertion sort.
3. The sort continues, putting 45, 71, and 88 in the correct order.
4. Only 1 of 3 interleaved lists has been sorted. 2 more InsertionSortInterleaved calls are needed, with a start index of 1 for the second list, and 2 for the third list.
5. Calling InsertionSortInterleaved with a starting index of 0 and a gap of 1 is equivalent to the regular insertion sort, and finishes sorting the list.

PARTICIPATION ACTIVITY 3.7.4: Insertion sort variant.

1) Given the call InsertionSortInterleaved(values, 10, 0, 5), what are the indices of the first two elements compared?

- ○ 1 and 5
- ○ 1 and 6
- ○ 0 and 4
- ○ 0 and 5

2) Given the call InsertionSortInterleaved(values, 4, 1, 4), what is the value of the loop variable i in the for loop's first iteration?

- ○ 1
- ○ 4
- ○ 5

3) InsertionSortInterleaved will result in an out of bounds array access if called on an array of size 4, a starting index of 1, and a gap value of 4.

- ○ True
- ○ False

4) If a gap value of 2 is chosen, then the following 2 function calls will fully sort a list:

InsertionSortInterleaved(list, 9, 0, 2)
InsertionSortInterleaved(list, 9, 1, 2)

- ○ True
- ○ False

Shell sort algorithm

Shell sort begins by picking an arbitrary collection of gap values. For each gap value K, K calls are made to the insertion sort variant function to sort K interleaved lists. Shell sort ends with a final gap value of 1, to finish with the regular insertion sort.

Shell sort tends to perform well when choosing gap values in descending order. A common option is to choose powers of 2 minus 1, in descending order. Ex: For an array of size 100, gap values would be 63, 31, 15, 7, 3, and 1. This gap selection technique results in shell sort's time complexity being no worse than $O(N^{3/2})$.

Using gap values that are powers of 2 or in descending order is not required. Shell sort will correctly sort arrays using any positive integer gap values in any order, provided a gap value of 1 is included.

PARTICIPATION ACTIVITY | 3.7.5: Shell sort algorithm.

Animation content:

`undefined`

Animation captions:

1. The first gap value of 5 causes 5 interleaved lists to be sorted. The inner for loop iterates over the start indices for the interleaved list. So, i is initialized with the first list's starting index, or 0.
2. The second for loop iteration sorts the interleaved list starting at index 1 with the same gap value of 5.
3. For the gap value 5, the remaining interleaved lists at start indices 2, 3, and 4 are sorted.
4. The next gap value of 3 causes 3 interleaved lists to be sorted. Few swaps are needed because the list is already partially sorted.
5. The final gap value of 1 finishes sorting.

PARTICIPATION ACTIVITY | 3.7.6: ShellSort.

1) ShellSort will properly sort an array using any collection of gap values, provided the collection contains 1.

○ True

○ False

2) Calling ShellSort with gap array (7, 3, 1) vs. (3, 7, 1) produces the same result with no difference in efficiency.

○ True

○ False

3) How many times is InsertionSortInterleaved called if ShellSort is called with gap array (10, 2, 1)?

○ 3

○ 12

○ 13

○ 20

CHALLENGE ACTIVITY 3.7.1: Shell sort.

361856.2187296.qx3zqy7

Start

Given a list (18, 13, 10, 86, 58, 21, 46, 93, 48) and a gap value of 2:

What is the first interleaved list?

(Ex: 1, 2, 3)
(comma between values)

What is the second interleaved list?

( )

1 2 3

Check Next

3.8 Java: Shell sort

Sorting interleaved arrays for shell sort

Shell sort uses a variation of the insertion sort algorithm to sort subsets of an input array. Instead of comparing elements that are immediately adjacent to each other, the insertion sort variant compares elements that are at a fixed distance apart, known as a gap space. Shell sort repeats this variation of insertion sort using different gap sizes and starting points within the input array.

Figure 3.8.1: Interleaved insertion sort algorithm.

```
void insertionSortInterleaved(int[] numbers, int startIndex, int gap) {
   for (int i = startIndex + gap; i < numbers.length; i += gap) {
      int j = i;
      while (j - gap >= startIndex && numbers[j] < numbers[j - gap]) {
         // Swap numbers[j] and numbers [j - gap]
         int temp = numbers[j];
         numbers[j] = numbers[j - gap];
         numbers[j - gap] = temp;
         j -= gap;
      }
   }
}
```

PARTICIPATION ACTIVITY 3.8.1: Sorting interleaved arrays.

1) insertionSort() is the same as insertionSortInterleaved() with a gap space of 1.

- True
- False

2) If the insertionSortInterleaved() method runs with an array of size 10, startIndex assigned with 2, and gap assigned with 3, then the loop variable i will be assigned with the values 2, 5, and 8.

- True

○ False

Shell sort algorithm

The shellSort() method calls the insertionSortInterleaved() method repeatedly using different gap sizes and start indices. Ex: If a gapValue is 3, then shellSort() will execute:

- `insertionSortInterleaved(numbers, 0, 3)`
- `insertionSortInterleaved(numbers, 1, 3)`
- `insertionSortInterleaved(numbers, 2, 3)`

All values from zero to gap - 1 are used as startIndex. This process repeats for all gap values. The shellSort() method takes as parameters the array to be sorted, and the array of gap values to be used.

Figure 3.8.2: Shell sort algorithm in Java.

```
void shellSort(int[] numbers, int[] gapValues) {
   for (int g = 0; g < gapValues.length; g++) {
      for (int i = 0; i < gapValues[g]; i++) {
         insertionSortInterleaved(numbers, i, gapValues[g]);
      }
   }
}
```

Choosing good gap values will minimize the total number of swap operations. The only requirement for the shell sort algorithm to work is that one of the gap values (usually the last one) is 1. Gap values are often chosen in decreasing order. A gap value of 1 is equivalent to the regular insertion sort algorithm. Thus, if larger gap values are previously used, the final insertion sort will do less work than if the regular insertion sort is done throughout.

PARTICIPATION ACTIVITY 3.8.2: Shell sort.

1) Which is a reasonable choice for gap values, given an input array of size 75?

○ 1, 3, 7, 15, 31

○ 31, 15, 7, 3, 1

○ 1, 2, 3

2) If shellSort() is run with an input array of size 20 and a gap values array of

{15, 7, 3, 1}, how many times will insertionSortInterleaved() be called?

- ○ 80
- ○ 4
- ○ 26

zyDE 3.8.1: Shell sort algorithm.

The following program uses shell sort to sort an array of integers. The algorithm has be
modified to print the array after sorting with each gap value and to return the total numb
swap operations used. Experiment with different number arrays and gap value arrays, a
compare the work done to the regular insertion sort algorithm.

ShellSortDemo.java Load default templ

```
import java.util.Arrays;

public class ShellSortDemo {
   private static int insertionSortInterleaved(int[] numbers, int startIn
      int swaps = 0;
      for (int i = startIndex + gap; i < numbers.length; i += gap) {
         int j = i;
         while (j - gap >= startIndex && numbers[j] < numbers[j - gap]) {
            swaps++;
            // Swap numbers[j] and numbers [j - 1]
            int temp = numbers[j];
            numbers[j] = numbers[j - gap];
            numbers[j - gap] = temp;
            j -= gap;
         }
      }
      return swaps;
```

Run