3.1 Sorting: Introduction

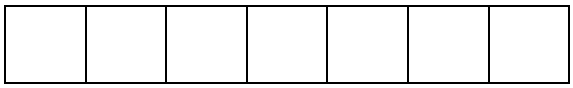
Sorting is the process of converting a list of elements into ascending (or descending) order. For example, given a list of numbers (17, 3, 44, 6, 9), the list after sorting is (3, 6, 9, 17, 44). You may have carried out sorting when arranging papers in alphabetical order, or arranging envelopes to have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

Start



Swap

Time - Best time -

Clear best

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 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 3.2.1: 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 3.2.2: 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 0 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 3.2.1: 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). 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 — times. So the total number of comparisons is proportional to —, or O(N). Other sorting algorithms involve more complex algorithms but have faster execution times.

PARTICIPATION ACTIVITY 3.2.3: Selection sort runtime.

Enter an integer value for each answer.

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

Check Show answer

2) About how many times longer will sorting a list of 2X elements take compared to sorting a list of X elements?

Check Show answer

3) About how many times longer will sorting a list of 10X elements take compared to sorting a list of X elements?

Check Show answer

CHALLENGE ACTIVITY 3.2.1: Selection sort.

502696.2096894.qx3zqy7

Start

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

Ex: 4

1 2 3 4

Check Next

3.3 Python: Selection sort

Selection sort

The selection_sort() function takes the list as a parameter, and has no return value since the list 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 list. The inner loop uses the variable j to examine all indices from i+1 to the end of the list. When the j loop finishes, the variable index_smallest will store the index position of the smallest item in the list from i onward. The final step is to swap the values at position i and index_smallest.

Figure 3.3.1: Selection sort.

```
def selection_sort(numbers):
   for i in range(len(numbers)-1):

      # Find index of smallest remaining element
      index_smallest = i
      for j in range(i+1, len(numbers)):

         if numbers[j] <
numbers[index_smallest]:
            index_smallest = j

      # Swap numbers[i] and
numbers[index_smallest]
      temp = numbers[i]
      numbers[i] = numbers[index_smallest]
      numbers[index_smallest] = temp
```


Figure 3.3.2: Python program to test selection sort.

```
# Create a list of numbers to sort
numbers = [10, 2, 78, 4, 45, 32, 7, 11]

# Display the contents of the list
print('UNSORTED:', numbers)

# Call the selection_sort() function
selection_sort(numbers)

# Display the (sorted) contents of the list
print('SORTED:', numbers)
```

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


PARTICIPATION ACTIVITY 3.3.1: Selection sort algorithm.

Suppose the selection_sort() function is executed with the list [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 selection_sort() function?

```
temp = numbers[i]
numbers[i] =
numbers[index_smallest]
numbers[index_smallest] = temp
```

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

zyDE 3.3.1: Selection sort algorithm.

This program uses the selection sort algorithm to sort a list. The code has been modified also calculate how many item comparisons are done. You can try running the program w different lists to see how the total number of comparisons changes (or doesn't change).

main.py Load default template... Run

```
1  def selection_sort(numbers):
2      # A variable to hold the number
3      comparisons = 0
4
5      for i in range(len(numbers)-1):
6
7          # Find index of smallest rema
8          index_smallest = i
9          for j in range(i+1, len(numbe
10
11             comparisons = comparisons
12             if numbers[j] < numbers[in
13                 index_smallest = j
14
15         # Swap numbers[i] and numbers
```

3.4 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 3.4.1: 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 3.4.1: 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 3.4.2: 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 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). 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 — times. So the total number of comparisons is proportional to — , or O(N). Other sorting algorithms involve more complex algorithms but faster execution.

PARTICIPATION ACTIVITY 3.4.3: 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 3.4.4: 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 3.4.5: 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 3.4.6: 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 3.4.1: Insertion sort.

502696.2096894.qx3zqy7

Start

Given list (21, 22, 31, 34, 36, 28, 23, 40, 29), what is the value of i when the first swap executes?

Ex: 1

1	2	3	4	5

Check Next

3.5 Python: Insertion sort

Insertion sort algorithm

The insertion_sort() function has one parameter, numbers, which is an unsorted list of elements. A list is a mutable object, so changes to numbers inside the function will all affect any variable that references

that list.

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 sorted part that are larger. Once a smaller or equal element is found in sorted part, the current element has been inserted in the correct location and the while loop terminates.

Figure 3.5.1: Insertion sort algorithm.

```
def insertion_sort(numbers):
    for i in range(1, len(numbers)):
        j = i

        # Insert numbers[i] into sorted part
        # stopping once numbers[i] in correct position
        while j > 0 and 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 -= 1

# Create a list of unsorted values
numbers = [10, 2, 78, 4, 45, 32, 7, 11]

# Print unsorted list
print('UNSORTED:', numbers)

# Call the insertion_sort function
insertion_sort(numbers)

# Print sorted list
print('SORTED:', numbers)
```

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

PARTICIPATION ACTIVITY 3.5.1: 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 insertion_sort function?

- temp
- range
- numbers

3) Which variable is returned by the insertion_sort function?

- def
- numbers
- None

zyDE 3.5.1: Insertion sort algorithm.

main.py Load default templat

```
def insertion_sort(numbers):
    for i in range(1, len(numbers)):
        j = i

        # Insert numbers[i] into sorted part
        # stopping once numbers[i] in correct position
        while j > 0 and 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 -= 1

# Create a list of unsorted values
numbers = [10, 2, 78, 4, 45, 32, 7, 11]
```

Run

3.6 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 3.6.1: 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 3.6.2: 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
- 3
- 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 3.6.3: 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 3.6.4: Insertion sort variant.

For each call to `InsertionSortInterleaved(numbersArray, X, ...)`, assume that numbersArray is an array of length X. Each question can be answered without knowing the contents of numbersArray.

1) `InsertionSortInterleaved(numbersArray, 10, 0, 5)` is called. What are the indices of the first two elements compared?

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

2) `InsertionSortInterleaved(numbersArray, 4, 1, 4)` is called. What is the value of the loop variable i first initialized to?

- 1
- 4
- 5

3) `InsertionSortInterleaved(numbersArray, 4, 1, 4)` results in an out of bounds array access.

- True
- False

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

```
InsertionSortInterleaved(numbersArray,
9, 0, 2)
InsertionSortInterleaved(numbersArray,
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 .

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 3.6.5: 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 3.6.6: 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 3.6.1: Shell sort.

502696.2096894.qx3zqy7

Start

Given an array [89, 84, 82, 28, 42, 45, 59, 77, 21, 87] and a gap value of 3:

What is the first interleaved array?

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

What is the second interleaved array?

[ ]

1 2 3

Check Next

3.7 Python: Shell sort

Sorting interleaved lists for shell sort

Shell sort uses a variation of the insertion sort algorithm to sort subsets of an input list. 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 list.

Figure 3.7.1: Interleaved insertion sort algorithm.

```
def insertion_sort_interleaved(numbers, start_index, gap):
    for i in range(start_index + gap, len(numbers), gap):
        j = i
        while (j - gap >= start_index) and (numbers[j] < numbers[j -
gap]):
            temp = numbers[j]
            numbers[j] = numbers[j - gap]
            numbers[j - gap] = temp
            j = j - gap
```

PARTICIPATION ACTIVITY 3.7.1: Sorting interleaved lists.

1) insertion_sort() is the same as insertion_sort_interleaved() with a gap space of 1.
   - True
   - False

2) If the insertion_sort_interleaved() function runs with a list of size 10, start_index 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 shell_sort() function calls the insertion_sort_interleaved() function repeatedly using different gap sizes and start indices. Ex: If a gap_value is 3, then shell_sort() will execute:

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

All values from zero to gap - 1 are used as start_index. This process repeats for all gap values. The shell_sort() function takes as parameters the list to be sorted, and the list of gap values to be used.

Figure 3.7.2: Interleaved insertion sort algorithm in Python.

```
def shell_sort(numbers, gap_values):
    for gap_value in gap_values:
        for i in range(gap_value):
            insertion_sort_interleaved(numbers, i,
gap_value)
```

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 3.7.2: Shell sort.

1) Which is a reasonable choice for gap values, given an input list of size 75?
   - [1, 3, 7, 15, 31]
   - [31, 15, 7, 3, 1]
   - [1, 2, 3]

2) If shell_sort() is run with an input list of size 20 and a gap values list of [15, 7, 3, 1], how many times will insertion_sort_interleaved() be called?

- 80
- 4
- 26

zyDE 3.7.1: Shell sort algorithm.

The following program uses shell sort to sort a list of integers. The algorithm has been modified to return the total number of swap operations required. Experiment with differer number lists and gap value lists, and compare the work done to the regular insertion sort algorithm. (The regular insertion sort algorithm is equivalent to using [1] as gap_values.)

main.py Load default templat

```
def insertion_sort_interleaved(numbers, start_index, gap):
    swaps = 0
    for i in range(start_index + gap, len(numbers), gap):
        j = i
        while (j - gap >= start_index) and (numbers[j] < numbers[j -
            swaps += 1
            temp = numbers[j]
            numbers[j] = numbers[j - gap]
            numbers[j - gap] = temp
            j = j - gap
    return swaps


def shell_sort(numbers, gap_values):
    swaps = []
```

Run

3.8 Quicksort

Quicksort

Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts. To partition the input, quicksort chooses a pivot to divide the data into low and high parts. The ***pivot*** can be any value within the array being sorted, commonly the value of the middle array element. Ex: For the list (4, 34, 10, 25, 1), the middle element is located at index 2 (the middle of indices [0, 4]) and has a value of 10.

Once the pivot is chosen, the quicksort algorithm divides the array into two parts, referred to as the low partition and the high partition. All values in the low partition are less than or equal to the pivot value. All values in the high partition are greater than or equal to the pivot value. The values in each partition are not necessarily sorted. Ex: Partitioning (4, 34, 10, 25, 1) with a pivot value of 10 results in a low partition of (4, 1, 10) and a high partition of (25, 34). Values equal to the pivot may appear in either or both of the partitions.

PARTICIPATION ACTIVITY 3.8.1: Quicksort partitions data into a low partition with values ≤ pivot and a high partition with values ≥ pivot.

Animation content:

Step 1: Array is shown as [7, 4, 6, 18, 8]. Labels lowIndex and highIndex point to array indices 0 and 4, respectively. Code execution begins within the Partition function, and the calculation of the midpoint is shown as:
lowIndex + (highIndex - lowIndex) / 2
= 0 + (4 - 0) / 2
= 2
The pivot variable is then assigned with numbers[midpoint], or 6.

Step 2: The first nested while loop executes. Since the condition 7 < 6 is false, lowIndex is not incremented and remains 0.

Step 3: The second nested while loop executes. Conditions 6 < 8 and 6 < 18 are true, so highIndex is decremented twice to become 2. The condition 6 < 6 is false, so the second nested while loop then ends.

Step 4: Elements at indices 0 and 2 (lowIndex and highIndex) are swapped, yielding the array: [6, 4, 7, 18, 8].

Step 5: Execution continues, changing lowIndex and highIndex to 1. The next iteration of the outermost loop begins. lowIndex is incremented to 2, since 4 < 6. The condition 7 < 6 is false, so lowIndex is not incremented further. The second nested while loop does not decrement highIndex, since the condition 6 < 4 is false. The following if statement's condition of lowIndex >= highIndex is now true, so the variable done is assigned with true.

Step 6: The Partition() function's execution ends, returning highIndex's value of 1.

Animation captions:

1. The pivot value is the value of the middle element.
2. lowIndex is incremented until a value greater than or equal to the pivot is found.
3. highIndex is decremented until a value less than or equal to the pivot is found.
4. Elements at indices lowIndex and highIndex are swapped, moving those elements to the correct partitions.
5. The partition process repeats until indices lowIndex and highIndex reach or pass each other, indicating all elements have been partitioned.
6. Once partitioned, the algorithm returns highIndex, which is the highest index of the low partition. The partitions are not yet sorted.

Partitioning algorithm

The partitioning algorithm uses two index variables lowIndex and highIndex, initialized to the left and right sides of the current elements being sorted. As long as the value at index lowIndex is less than the pivot value, the algorithm increments lowIndex, because the element should remain in the low partition. Likewise, as long as the value at index highIndex is greater than the pivot value, the algorithm decrements highIndex, because the element should remain in the high partition. Then, if lowIndex >= highIndex, all elements have been partitioned, and the partitioning algorithm returns highIndex, which is the index of the last element in the low partition. Otherwise, the elements at indices lowIndex and highIndex are swapped to move those elements to the correct partitions. The algorithm then increments lowIndex, decrements highIndex, and repeats.

PARTICIPATION ACTIVITY 3.8.2: Quicksort pivot location and value.

Determine the midpoint and pivot values.

1) numbers = (1, 2, 3, 4, 5), lowIndex = 0, highIndex = 4

```
midpoint =
```

Check Show answer

2) numbers = (1, 2, 3, 4, 5), lowIndex = 0, highIndex = 4

```
pivot =
```

Check Show answer

3) numbers = (200, 11, 38, 9), lowIndex = 0, highIndex = 3

```
midpoint =
```

Check Show answer

4) numbers = (200, 11, 38, 9), lowIndex = 0, highIndex = 3

```
pivot =
```

Check Show answer

5) numbers = (55, 7, 81, 26, 0, 34, 68, 125), lowIndex = 3, highIndex = 7

```
midpoint =
```

Check Show answer

6) numbers = (55, 7, 81, 26, 0, 34, 68, 125), lowIndex = 3, highIndex = 7

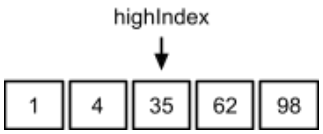
```
pivot =
```

Check Show answer

PARTICIPATION ACTIVITY 3.8.3: Low and high partitions.

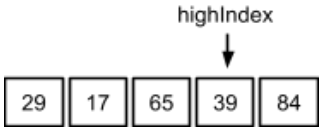
Determine if the low and high partitions are correct given highIndex and pivot.

1) pivot = 35



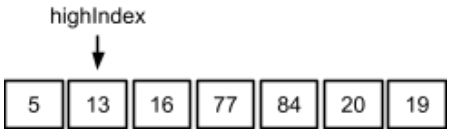
- Correct
- Incorrect

2) pivot = 65



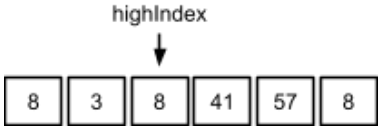
- Correct
- Incorrect

3) pivot = 5



- Correct
- Incorrect

4) pivot = 8



- Correct
- Incorrect

Recursively sorting partitions

Once partitioned, each partition needs to be sorted. Quicksort is typically implemented as a recursive algorithm using calls to quicksort to sort the low and high partitions. This recursive sorting process continues until a partition has one or zero elements, and thus is already sorted.

PARTICIPATION ACTIVITY 3.8.4: Quicksort.

Animation content:

undefined

Animation captions:

1. The list from low index 0 to high index 4 has more than 1 element, so Partition is called.
2. Quicksort is called recursively to sort the low and high partitions.
3. The low partition has more than one element. Partition is called for the low partition, followed by recursive calls to Quicksort.
4. Each partition that has one element is already sorted.
5. The high partition has more than one element and thus is partitioned and recursively sorted.
6. The low partition with two elements is partitioned and recursively sorted.

7. Each remaining partition with only one element is already sorted.
8. All elements are sorted.

Below is the recursive quicksort algorithm, including quicksort's key component, the partitioning function.

Figure 3.8.1: Quicksort algorithm.

©zyBooks 11/20/23 10:58 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
Partition(numbers, lowIndex, highIndex) {
   // Pick middle element as pivot
   midpoint = lowIndex + (highIndex - lowIndex) / 2
   pivot = numbers[midpoint]

   done = false
   while (!done) {
      // Increment lowIndex while numbers[lowIndex] < pivot
      while (numbers[lowIndex] < pivot) {
         lowIndex += 1
      }

      // Decrement highIndex while pivot < numbers[highIndex]
      while (pivot < numbers[highIndex]) {
         highIndex -= 1
      }

      // If zero or one elements remain, then all numbers are
      // partitioned. Return highIndex.
      if (lowIndex >= highIndex) {
         done = true
      }
      else {
         // Swap numbers[lowIndex] and numbers[highIndex]
         temp = numbers[lowIndex]
         numbers[lowIndex] = numbers[highIndex]
         numbers[highIndex] = temp

         // Update lowIndex and highIndex
         lowIndex += 1
         highIndex -= 1
      }
   }

   return highIndex
}

Quicksort(numbers, lowIndex, highIndex) {
   // Base case: If the partition size is 1 or zero
   // elements, then the partition is already sorted
   if (lowIndex >= highIndex) {
      return
   }

   // Partition the data within the array. Value lowEndIndex
   // returned from partitioning is the index of the low
   // partition's last element.
   lowEndIndex = Partition(numbers, lowIndex, highIndex)

   // Recursively sort low partition (lowIndex to lowEndIndex)
   // and high partition (lowEndIndex + 1 to highIndex)
   Quicksort(numbers, lowIndex, lowEndIndex)
   Quicksort(numbers, lowEndIndex + 1, highIndex)
}

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()

   // Initial call to quicksort
   Quicksort(numbers, 0, NUMBERS_SIZE - 1)

   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
```

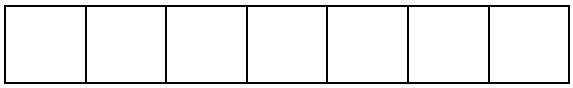
Quicksort activity

The following activity helps build intuition as to how partitioning a list into two unsorted parts, one part <= a pivot value and the other part >= a pivot value, and then recursively sorting each part, ultimately leads to a sorted list.

PARTICIPATION ACTIVITY 3.8.5: Quicksort tool.

Select all values in the current window that are less than the pivot for the left part, then press "Partition". If a value equals pivot, you can choose which part, but each part must contain at least one number. Light blue means current window. Green means sorted.

Start



Partition Back

Time - Best time -

Clear best

Quicksort runtime

The quicksort algorithm's runtime is typically O(N log N). Quicksort has several partitioning levels, the first level dividing the input into 2 parts, the second into 4 parts, the third into 8 parts, etc. At each level, the algorithm does at most N comparisons moving the lowIndex and highIndex indices. If the pivot yields two equal-sized parts, then there will be log N levels, requiring the N * log N comparisons.

PARTICIPATION ACTIVITY 3.8.6: Quicksort runtime.

Assume quicksort always chooses a pivot that divides the elements into two equal parts.

1) How many partitioning levels are required for a list of 8 elements?

Check Show answer

2) How many partitioning levels are required for a list of 1024 elements?

Check Show answer

3) How many total comparisons are required to sort a list of 1024 elements?

Check Show answer

Worst case runtime

For typical unsorted data, such equal partitioning occurs. However, partitioning may yield unequally sized parts in some cases. If the pivot selected for partitioning is the smallest or largest element, one partition will have just 1 element, and the other partition will have all other elements. If this unequal partitioning happens at every level, there will be N - 1 levels, yielding N + N-1 + N-2 + ... + 2 + 1 = , which is O(N). So the worst case runtime for the quicksort algorithm is O(N). Fortunately, this worst case runtime rarely occurs.

PARTICIPATION ACTIVITY 3.8.7: Worst case quicksort runtime.

Assume quicksort always chooses the smallest element as the pivot.

1) Given numbers = (7, 4, 2, 25, 19), lowIndex = 0, and highIndex = 4, what are the contents of the low partition? Type answer as: 1, 2, 3

Check Show answer

2) How many partitioning levels are required for a list of 5 elements?

Check Show answer

3) How many partitioning levels are required for a list of 1024 elements?

Check Show answer

4) How many total calls to the Quicksort() function are made to sort a list of 1024 elements?

Check Show answer

CHALLENGE ACTIVITY 3.8.1: Quicksort.

502696.2096894.qx3zqy7

Start

Given numbers = (52, 33, 94, 93, 22), lowIndex = 0, highIndex = 4

What is the midpoint? Ex: 9

What is the pivot?

1	2	3	4	5	6

Check Next

3.9 Python: Quicksort

The partition() function

The quicksort algorithm works by partitioning a section of the unsorted list into a left part and a right part, based on a chosen element within the list called the pivot. The partition() function has three parameters: the unsorted list, the start index, and the end index. When the function completes, the

elements between the start and end indices are reorganized so that the elements in the left part are less than or equal to the pivot and the elements in the right part are greater than or equal to the pivot. The left and right parts may be different sizes, and the method returns the index of the last item in the left part. The left and right parts are not, themselves, sorted.

Figure 3.9.1: The partition() function used by quicksort.


```
def partition(numbers, start_index, end_index):
    # Select the middle value as the pivot.
    midpoint = start_index + (end_index - start_index)
// 2
    pivot = numbers[midpoint]

    # "low" and "high" start at the ends of the list
segment
    # and move towards each other.
    low = start_index
    high = end_index

    done = False
    while not done:
        # Increment low while numbers[low] < pivot
        while numbers[low] < pivot:
            low = low + 1

        # Decrement high while pivot < numbers[high]
        while pivot < numbers[high]:
            high = high - 1

        # If low and high have crossed each other, the
loop
        # is done. If not, the elements are swapped, low
is
        # incremented and high is decremented.
        if low >= high:
            done = True
        else:
            temp = numbers[low]
            numbers[low] = numbers[high]
            numbers[high] = temp
            low = low + 1
            high = high - 1

    # "high" is the last index in the left segment.
    return high
```


PARTICIPATION ACTIVITY 3.9.1: List partitioning.

1) The value 15 would be selected as the pivot in the list:
[5, 3, 15, 72, 14, 41, 32, 18].

- True
- False

2) After partitioning, the left and right parts should be sorted.

- True
- False

3) The following list is properly partitioned with pivot 17:
[3, 1, 14, 12, 19, 17, 22]

- True
- False

4) The partition method has runtime complexity.

- True
- False

The quicksort() algorithm

The quicksort() function uses recursion to sort the two parts of the list, thus sorting the full list. The function has three parameters: the unsorted list, the start index, and the end index. quicksort() starts by calling partition() to partition the list into left (low) and right(high) parts. quicksort() then calls itself, using recursion to sort the two list parts.

A program can sort a list by calling quicksort() and specifying start_index as 0 and end_index as the index of the last item in the list.

Figure 3.9.2: The quicksort() algorithm.

```
def quicksort(numbers, start_index, end_index):
    # Only attempt to sort the list segment if there
are
    # at least 2 elements
    if end_index <= start_index:
        return

    # Partition the list segment
    high = partition(numbers, start_index, end_index)

    # Recursively sort the left segment
    quicksort(numbers, start_index, high)

    # Recursively sort the right segment
    quicksort(numbers, high + 1, end_index)
```


Figure 3.9.3: A program to run quicksort.

```
# Main program to test the quicksort algorithm.
numbers = [12, 18, 3, 7, 32, 14, 91, 16, 8, 57]
print('UNSORTED:', numbers)

quicksort(numbers, 0, len(numbers)-1)
print('SORTED:', numbers)
```

```
UNSORTED: [12, 18, 3, 7, 32, 14, 91, 16, 8, 57]
SORTED: [3, 7, 8, 12, 14, 16, 18, 32, 57, 91]
```

PARTICIPATION ACTIVITY 3.9.2: The quicksort algorithm.

1) quicksort()'s third parameter is the list's length.

   - True
   - False

2) Suppose the following code is executed:

```
numbers = [8, 2, 7, 6, 1, 4, 3,
5]
quicksort(numbers, 3, 6)
```

After this code finishes, the numbers list would be:
[8, 2, 7, 1, 3, 4, 6, 5].

- True
- False

zyDE 3.9.1: Quicksort algorithm.

The following program includes the partition() and quicksort() functions. Run the program different lists and different parameters for the quicksort() function to gain familiarity with algorithm.

main.py Load default templat

```
1  def partition(numbers, start_index, end_index):
2      # Select the middle value as the pivot.
3      midpoint = start_index + (end_index - start_index) // 2
4      pivot = numbers[midpoint]
5
6      # "low" and "high" start at the ends of the list segment
7      # and move towards each other.
8      low = start_index
9      high = end_index
10
11     done = False
12     while not done:
13         # Increment low while numbers[low] < pivot
14         while numbers[low] < pivot:
15             low = low + 1
```

Run

3.10 Merge sort

Merge sort overview

Merge sort is a sorting algorithm that divides a list into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted list. The recursive partitioning continues until a list of 1 element is reached, as a list of 1 element is already sorted.

PARTICIPATION ACTIVITY 3.10.1: Merge sort recursively divides the input into two halves, sorts each half, and merges the lists together.

Animation captions:

1. MergeSort recursively divides the list into two halves.
2. The list is divided until a list of 1 element is found.
3. A list of 1 element is already sorted.
4. At each level, the sorted lists are merged together while maintaining the sorted order.

Merge sort partitioning

The merge sort algorithm uses three index variables to keep track of the elements to sort for each recursive function call. The index variable i is the index of first element in the list, and the index variable k is the index of the last element. The index variable j is used to divide the list into two halves. Elements from i to j are in the left half, and elements from j + 1 to k are in the right half.

PARTICIPATION ACTIVITY 3.10.2: Merge sort partitioning.

Determine the index j and the left and right partitions.

1) numbers = (1, 2, 3, 4, 5), i = 0, k = 4

```
j =
```

Check Show answer

2) numbers = (1, 2, 3, 4, 5), i = 0, k = 4

```
Left partition  = (
                  )
```

Check Show answer

3) numbers = (1, 2, 3, 4, 5), i = 0, k = 4

```
Right partition  = (
                   )
```

Check Show answer

4) numbers = (34, 78, 14, 23, 8, 35), i = 3, k = 5

```
j =
```

Check Show answer

5) numbers = (34, 78, 14, 23, 8, 35), i = 3, k = 5

```
Left partition  = (
                  )
```

Check Show answer

6) numbers = (34, 78, 14, 23, 8, 35), i = 3, k = 5

```
Right partition  = (
                   )
```

Check Show answer

Merge sort algorithm

Merge sort merges the two sorted partitions into a single list by repeatedly selecting the smallest element from either the left or right partition and adding that element to a temporary merged list. Once fully merged, the elements in the temporary merged list are copied back to the original list.

PARTICIPATION ACTIVITY 3.10.3: Merging partitions: Smallest element from left or right partition is added one at a time to a temporary merged list. Once merged, temporary list is copied back to the original list.

Animation content:

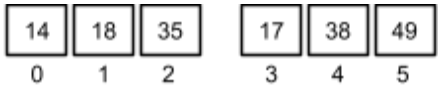
undefined

Animation captions:

1. Create a temporary list for merged numbers. Initialize mergePos, leftPos, and rightPos to the first element of each of the corresponding list.
2. Compare the element in the left and right partitions. Add the smallest value to the temporary list and update the relevant indices.
3. Continue to compare the elements in the left and right partitions until one of the partitions is empty.
4. If a partition is not empty, copy the remaining elements to the temporary list. The elements are already in sorted order.
5. Lastly, the elements in the temporary list are copied back to the original list.

PARTICIPATION ACTIVITY 3.10.4: Tracing merge operation.

Trace the merge operation by determining the next value added to mergedNumbers.



1) leftPos = 0, rightPos = 3

Check Show answer

2) leftPos = 1, rightPos = 3

Check Show answer

3) leftPos = 1, rightPos = 4

Check Show answer

4) leftPos = 2, rightPos = 4

Check Show answer

5) leftPos = 3, rightPos = 4

Check Show answer

6) leftPos = 3, rightPos = 5

Check Show answer

Figure 3.10.1: Merge sort algorithm.

©zyBooks 11/20/23 10:58 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
Merge(numbers, i, j, k) {
   mergedSize = k - i + 1               // Size of merged partition
   mergePos = 0                         // Position to insert merged number
   leftPos = 0                          // Position of elements in left
partition
   rightPos = 0                         // Position of elements in right
partition
   mergedNumbers = new int[mergedSize]  // Dynamically allocates temporary
array
                                        // for merged numbers

   leftPos = i                          // Initialize left partition
position
   rightPos = j + 1                     // Initialize right partition
position

   // Add smallest element from left or right partition to merged numbers
   while (leftPos <= j && rightPos <= k) {
      if (numbers[leftPos] <= numbers[rightPos]) {
         mergedNumbers[mergePos] = numbers[leftPos]
         ++leftPos
      }
      else {
         mergedNumbers[mergePos] = numbers[rightPos]
         ++rightPos

      }
      ++mergePos
   }

   // If left partition is not empty, add remaining elements to merged
numbers
   while (leftPos <= j) {
      mergedNumbers[mergePos] = numbers[leftPos]
      ++leftPos
      ++mergePos
   }

   // If right partition is not empty, add remaining elements to merged
numbers
   while (rightPos <= k) {
      mergedNumbers[mergePos] = numbers[rightPos]
      ++rightPos
      ++mergePos
   }

   // Copy merge number back to numbers
   for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
      numbers[i + mergePos] = mergedNumbers[mergePos]
   }
}

MergeSort(numbers, i, k) {
   j = 0

   if (i < k) {
      j = (i + k) / 2  // Find the midpoint in the partition

      // Recursively sort left and right partitions
      MergeSort(numbers, i, j)
```

```
      MergeSort(numbers, j + 1, k)

      // Merge left and right partition in sorted order
      Merge(numbers, i, j, k)
   }
}

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()

   MergeSort(numbers, 0, NUMBERS_SIZE - 1)

   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
```

Merge sort runtime

The merge sort algorithm's runtime is O(N log N). Merge sort divides the input in half until a list of 1 element is reached, which requires log N partitioning levels. At each level, the algorithm does about N comparisons selecting and copying elements from the left and right partitions, yielding N * log N comparisons.

Merge sort requires O(N) additional memory elements for the temporary array of merged elements. For the final merge operation, the temporary list has the same number of elements as the input. Some sorting algorithms sort the list elements in place and require no additional memory, but are more complex to write and understand.

To allocate the temporary array, the `Merge()` function dynamically allocates the array. mergedNumbers is a pointer variable that points to the dynamically allocated array, and `new int[mergedSize]` allocates the array with mergedSize elements. Alternatively, instead of allocating the array within the `Merge()` function, a temporary array with the same size as the array being sorted can be passed as an argument.

PARTICIPATION ACTIVITY 3.10.5: Merge sort runtime and memory complexity.

1) How many recursive partitioning levels are required for a list of 8 elements?

Check Show answer

2) How many recursive partitioning levels are required for a list of 2048 elements?

Check Show answer

3) How many elements will the temporary merge list have for merging two partitions with 250 elements each?

Check Show answer

CHALLENGE ACTIVITY 3.10.1: Merge sort.

502696.2096894.qx3zqy7

Start

numbers:

38	66	50	61	39	98

What call sorts the numbers array?

MergeSort(numbers, Ex: 1 ,)

Check Next

3.11 Python: Merge sort

Merge sort algorithm

Merge sort is a sorting algorithm that divides a list into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted list. Merge sort uses 2 functions: merge() and merge_sort(). The merge() function merges 2 sequential, sorted partitions within a list and has 4 parameters:

1. The list of numbers containing the 2 sorted partitions to merge
2. The start index of the first sorted partition
3. The end index of the first sorted partition
4. The end index of the second sorted partition

The merge_sort() function sorts a partition in a list and has 3 parameters:

1. The list containing the partition to sort
2. The start index of the partition to sort
3. The end index of the partition to sort

If the partition size is greater than 1, the merge_sort() function recursively sorts the left and right halves of the partition, then merges the sorted halves together. When the start index is 0 and the end index is the list length minus 1, merge_sort() sorts the entire list.

Figure 3.11.1: Merge sort algorithm.

©zyBooks 11/20/23 10:58 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
def merge(numbers, i, j, k):
    merged_size = k - i + 1                # Size of merged partition
    merged_numbers = [0] * merged_size     # Dynamically allocates temporary
array
                                           # for merged numbers
    merge_pos = 0                          # Position to insert merged number
    left_pos = i                           # Initialize left partition
position
    right_pos = j + 1                      # Initialize right partition
position

    # Add smallest element from left or right partition to merged numbers
    while left_pos <= j and right_pos <= k:
        if numbers[left_pos] <= numbers[right_pos]:
            merged_numbers[merge_pos] = numbers[left_pos]
            left_pos += 1
        else:
            merged_numbers[merge_pos] = numbers[right_pos]
            right_pos += 1
        merge_pos = merge_pos + 1

    # If left partition is not empty, add remaining elements to merged
numbers
    while left_pos <= j:
        merged_numbers[merge_pos] = numbers[left_pos]
        left_pos += 1
        merge_pos += 1

    # If right partition is not empty, add remaining elements to merged
numbers
    while right_pos <= k:
        merged_numbers[merge_pos] = numbers[right_pos]
        right_pos = right_pos + 1
        merge_pos = merge_pos + 1

    # Copy merge number back to numbers
    for merge_pos in range(merged_size):
        numbers[i + merge_pos] = merged_numbers[merge_pos]


def merge_sort(numbers, i, k):
    j = 0

    if i < k:
        j = (i + k) // 2  # Find the midpoint in the partition

        # Recursively sort left and right partitions
        merge_sort(numbers, i, j)
        merge_sort(numbers, j + 1, k)

        # Merge left and right partition in sorted order
        merge(numbers, i, j, k)


# Create a list of unsorted values
numbers = [61, 76, 19, 4, 94, 32, 27, 83, 58]

# Print unsorted list
print('UNSORTED:', numbers)
```

```
# Initial call to merge_sort
merge_sort(numbers, 0, len(numbers) - 1)

# Print sorted list
print('SORTED:', numbers)
```

```
UNSORTED: [61, 76, 19, 4, 94, 32, 27, 83, 58]
SORTED: [4, 19, 27, 32, 58, 61, 76, 83, 94]
```


PARTICIPATION ACTIVITY 3.11.1: Merge sort in Python.

1) To sort a list with 10 elements, the arguments passed to merge_sort will be: list, 0, 10.
   - True
   - False

2) If merge_sort is called with i = 0 and k = 4, what is the size of the partition that will be sorted?
   - 3
   - 4
   - 5

3) If merge_sort is called with i = 0 and k = 5, what is the value of j after the following line executes?
   `j = (i + k) // 2`
   - 2
   - 2.5

zyDE 3.11.1: Merge sort in Python.

main.py

Load default templat

```
1 def merge(numbers, i, j, k):
2     merged_size = k - i + 1                # Size of merged partition
3     merged_numbers = [0] * merged_size     # Dynamically allocates tem
4                                            # for merged numbers
5     merge_pos = 0                          # Position to insert merged
6     left_pos = i                           # Initialize left partition
7     right_pos = j + 1                      # Initialize right partitio
```

```
8
9       # Add smallest element from left or right partition to merged num
10      while left_pos <= j and right_pos <= k:
11          if numbers[left_pos] <= numbers[right_pos]:
12              merged_numbers[merge_pos] = numbers[left_pos]
13              left_pos += 1
14          else:
15              merged_numbers[merge_pos] = numbers[right_pos]
```


Run

3.12 Radix sort

Buckets

Radix sort is a sorting algorithm designed specifically for integers. The algorithm makes use of a concept called buckets and is a type of bucket sort.

Any array of integer values can be subdivided into buckets by using the integer values' digits. A ***bucket*** is a collection of integer values that all share a particular digit value. Ex: Values 57, 97, 77, and 17 all have a 7 as the 1's digit, and would all be placed into bucket 7 when subdividing by the 1's digit.

PARTICIPATION ACTIVITY 3.12.1: A particular single digit in an integer can determine the integer's bucket.

Animation captions:

1. Using only the 1's digit, each integer can be put into a bucket. 736 is put into bucket 6, 81 into bucket 1, 101 into bucket 1, and so on.
2. Using only the 10's digit, each integer can be put into a bucket. 736 is put into bucket 3, 81 into bucket 8, and so on. 5 is like 05 so is put into bucket 0.
3. Using only the 100's digit, each integer can be put into a bucket. 736 is put into bucket 7, 81 is like 081 so is put into bucket 0, and so on.

PARTICIPATION ACTIVITY 3.12.2: Using the 1's digit, place each integer in the correct bucket.

If unable to drag and drop, refresh the page.

7 49 74 50

	Bucket 0
	Bucket 4
	Bucket 7
	Bucket 9

Reset

PARTICIPATION ACTIVITY 3.12.3: Using the 10's digit, place each integer in the correct bucket.

If unable to drag and drop, refresh the page.

50 7 74 86

	Bucket 0
	Bucket 5
	Bucket 7
	Bucket 8

Reset

PARTICIPATION ACTIVITY 3.12.4: Bucket concepts.

1) Integers will be placed into buckets based on the 1's digit. More buckets are needed for an array with one thousand integers than for an array with one hundred integers.

- ○ True
- ○ False

2) Consider integers X and Y, such that X < Y. X will always be in a lower bucket than Y.

- ○ True
- ○ False

3) All integers from an array could be placed into the same bucket, even if the array has no duplicates.

- ○ True
- ○ False

PARTICIPATION ACTIVITY 3.12.5: Assigning integers to buckets.

For each question, consider the array of integers: 51, 47, 96, 52, 27.

1) When placing integers using the 1's digit, how many integers will be in bucket 7?

- ○ 0
- ○ 1
- ○ 2

2) When placing integers using the 1's digit, how many integers will be in bucket 5?

- ○ 0
- ○ 1
- ○ 2

3) When placing integers using the 10's digit, how many will be in bucket 9?

- ○ 0
- ○ 1
- ○ 2

4) All integers would be in bucket 0 if using the 100's digit.

- ○ True
- ○ False

Radix sort algorithm

Radix sort is a sorting algorithm specifically for an array of *integers*: The algorithm processes one digit at a time starting with the least significant digit and ending with the most significant. Two steps are needed for each digit. First, all array elements are placed into buckets based on the current digit's value. Then, the array is rebuilt by removing all elements from buckets, in order from lowest bucket to highest.

PARTICIPATION ACTIVITY 3.12.6: Radix sort algorithm (for non-negative integers).

Animation content:

undefined

Animation captions:

1. Radix sort begins by allocating 10 buckets and putting each number in a bucket based on the 1's digit.
2. Numbers are taken out of buckets, in order from lowest bucket to highest, rebuilding the array.
3. The process is repeated for the 10's digit. First, the array numbers are placed into buckets based on the 10's digit.
4. The items are copied from buckets back into the array. Since all digits have been processed, the result is a sorted array.

Figure 3.12.1: RadixGetMaxLength and RadixGetLength functions.

©zyBooks 11/20/23 10:58 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
// Returns the maximum length, in number of digits, out of all elements in
the array
RadixGetMaxLength(array, arraySize) {
   maxDigits = 0
   for (i = 0; i < arraySize; i++) {
      digitCount = RadixGetLength(array[i])
      if (digitCount > maxDigits)
         maxDigits = digitCount
   }
   return maxDigits
}

// Returns the length, in number of digits, of value
RadixGetLength(value) {
   if (value == 0)
      return 1

   digits = 0
   while (value != 0) {
      digits = digits + 1
      value = value / 10
   }
   return digits
}
```


PARTICIPATION ACTIVITY 3.12.7: Radix sort algorithm.

1) What will RadixGetLength(17) evaluate to?

Check Show answer

2) What will RadixGetMaxLength return when the array is (17, 4, 101)?

Check Show answer

3) When sorting the array (57, 5, 501) with RadixSort, what is the largest number of integers that will be in bucket 5 at any given moment?

Check Show answer

PARTICIPATION ACTIVITY 3.12.8: Radix sort algorithm analysis.

1) When sorting an array of n 3-digit integers, RadixSort's worst-case time complexity is O(n).
   - True
   - False

2) When sorting an array with n elements, the maximum number of elements that RadixSort may put in a bucket is n.
   - True
   - False

3) RadixSort has a space complexity of O(1).
   - True
   - False

4) The RadixSort() function shown above also works on an array of floating-point values.
   - True
   - False

Sorting signed integers

The above radix sort algorithm correctly sorts arrays of non-negative integers. But if the array contains negative integers, the above algorithm would sort by absolute value, so the integers are not correctly sorted. A small extension to the algorithm correctly handles negative integers.

In the extension, before radix sort completes, the algorithm allocates two buckets, one for negative integers and the other for non-negative integers. The algorithm iterates through the array in order, placing negative integers in the negative bucket and non-negative integers in the non-negative bucket. The algorithm then reverses the order of the negative bucket and concatenates the buckets to yield a sorted array. Pseudocode for the completed radix sort algorithm follows.

Figure 3.12.2: RadixSort algorithm (for negative and non-negative integers).


```
RadixSort(array, arraySize) {
   buckets = create array of 10 buckets

   // Find the max length, in number of digits
   maxDigits = RadixGetMaxLength(array, arraySize)

   pow10 = 1
   for (digitIndex = 0; digitIndex < maxDigits;
digitIndex++) {
      for (i = 0; i < arraySize; i++) {
         bucketIndex = abs(array[i] / pow10) % 10
         Append array[i] to buckets[bucketIndex]
      }
      arrayIndex = 0
      for (i = 0; i < 10; i++) {
         for (j = 0; j < buckets[i]→size(); j++) {
            array[arrayIndex] = buckets[i][j]
            arrayIndex = arrayIndex + 1
         }
      }
      pow10 = pow10 * 10
      Clear all buckets
   }

   negatives = all negative values from array
   nonNegatives = all non-negative values from array
   Reverse order of negatives
   Concatenate negatives and nonNegatives into array
}
```

PARTICIPATION ACTIVITY 3.12.9: Sorting signed integers.

For each question, assume radix sort has sorted integers by absolute value to produce the array (-12, 23, -42, 73, -78), and is about to build the negative and non-negative buckets to complete the sort.

1) What integers will be placed into the negative bucket? Type answer as: 15, 42, 98

( )

Check Show answer

2) What integers will be placed into the non-negative bucket? Type answer as: 15, 42, 98

( )

Check Show answer

3) After reversal, what integers are in the negative bucket? Type answer as: 15, 42, 98

( )

Check Show answer

4) What is the final array after RadixSort concatenates the two buckets? Type answer as: 15, 42, 98

( )

Check Show answer

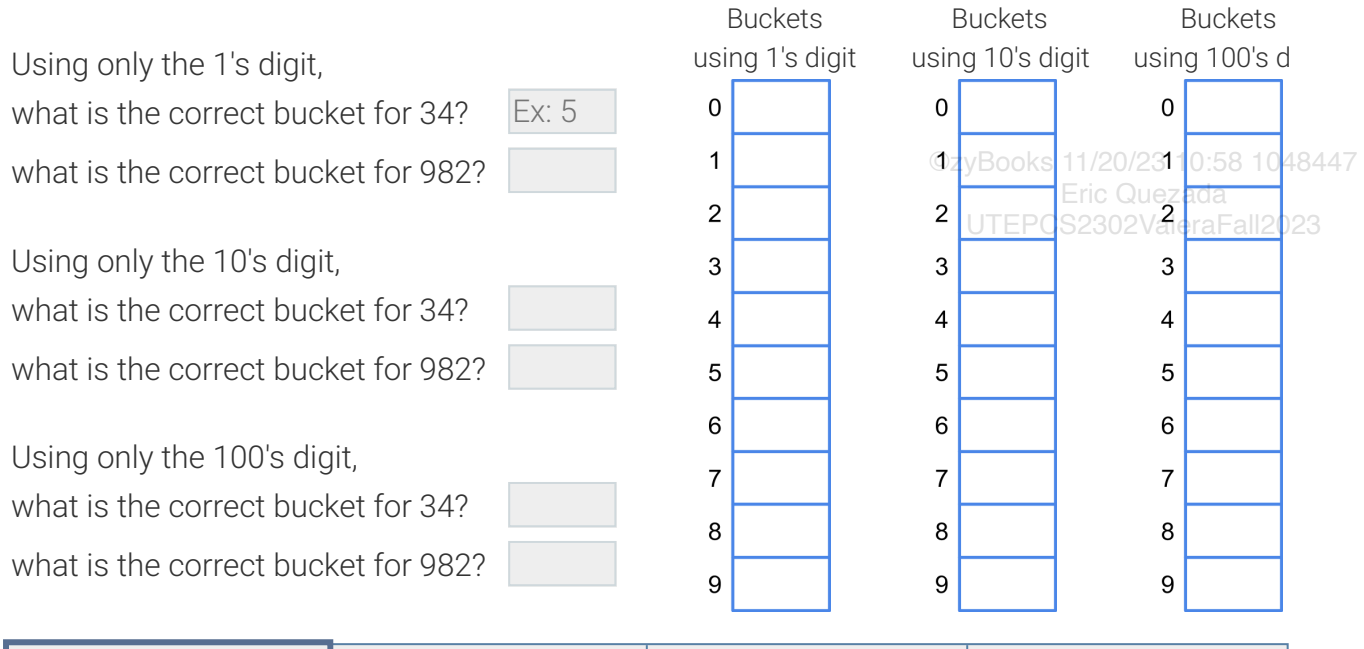
Radix sort with different bases

This section presents radix sort with base 10, but other bases can be used as well. Ex: Using base 2 is another common approach, where only 2 buckets would be required, instead of 10.

CHALLENGE ACTIVITY 3.12.1: Radix sort.

502696.2096894.qx3zqy7

Start



1 2 3 4

Check Next

3.13 Python: Radix sort

Radix sort algorithm

The ***radix sort*** algorithm sorts a list of integers by grouping elements based on the element's digits, starting with the least significant digit and ending with the most significant. Two steps are needed for each digit. First, all list elements are placed into buckets based on the current digit's value. Then, the list is rebuilt by removing all elements from buckets, in order from lowest bucket to highest.

The radix_sort() function below has one parameter, numbers, which is an unsorted list of integers. A list is a mutable object, so changes to numbers inside the function will all affect any variable that references that list.

A list of 10 lists is used for the buckets. The radix_get_max_length function determines the number of relevant powers of 10. The digit-index loop then iterates through those powers of 10. The first inner loop places the elements into buckets based on the current power of 10. The second inner loop moves elements out of buckets and back into the numbers list, in order from lowest bucket index to highest. After the digit-index loop, two new lists are built, one containing all negative elements from the numbers

list and the other containing all non-negative elements. Then, the reversed negative list is concatenated with the non-negative list into the numbers list to produce the sorted result.

Figure 3.13.1: Radix sort algorithm.

©zyBooks 11/20/23 10:58 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

©zyBooks 11/20/23 10:58 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
# Returns the maximum length, in number of digits, out of all list elements
def radix_get_max_length(numbers):
    max_digits = 0
    for num in numbers:
        digit_count = radix_get_length(num)
        if digit_count > max_digits:
            max_digits = digit_count
    return max_digits


# Returns the length, in number of digits, of value
def radix_get_length(value):
    if value == 0:
        return 1

    digits = 0
    while value != 0:
        digits += 1
        value = int(value / 10)
    return digits


def radix_sort(numbers):
    buckets = []
    for i in range(10):
        buckets.append([])

    # Find the max length, in number of digits
    max_digits = radix_get_max_length(numbers)

    pow_10 = 1
    for digit_index in range(max_digits):
        for num in numbers:
            bucket_index = (abs(num) // pow_10) % 10
            buckets[bucket_index].append(num)

        numbers.clear()
        for bucket in buckets:
            numbers.extend(bucket)
            bucket.clear()

        pow_10 = pow_10 * 10

    negatives = []
    non_negatives = []
    for num in numbers:
        if num < 0:
            negatives.append(num)
        else:
            non_negatives.append(num)
    negatives.reverse()
    numbers.clear()
    numbers.extend(negatives + non_negatives)


# Create a list of unsorted values
numbers = [47, 81, 13, 5, 38, 96, 51, 64]

# Print unsorted list
print('UNSORTED:', numbers)
```


©zyBooks 11/20/23 10:58 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

```
# Call radix_sort to sort the list
radix_sort(numbers)

# Print sorted list
print('SORTED:', numbers)
```

```
UNSORTED: [47, 81, 13, 5, 38, 96, 51, 64]
SORTED: [5, 13, 38, 47, 51, 64, 81, 96]
```


PARTICIPATION ACTIVITY 3.13.1: Radix sort algorithm implementation.

1) If the list [56, 19, 2, 101, 70] is passed to radix_sort(), after the line `max_digits = radix_get_max_length(numbers)`, what is the value of max_digits?
   - 1
   - 3
   - 5

2) If the numbers list is empty prior to the line `numbers.extend(bucket)`, which expression represents the length of numbers after this line?
   - 1
   - len(bucket)

3) The radix_sort() function would still operate correctly if the last line is changed from `numbers.extend(negatives + non_negatives)` to `numbers = negatives + non_negatives`.
   - True
   - False

zyDE 3.13.1: Radix sort in Python.

main.py Load default templat

```
# Returns the maximum length, in number of digits, out of all list el
def radix_get_max_length(numbers):
    max_digits = 0
    for num in numbers:
        digit_count = radix_get_length(num)
        if digit_count > max_digits:
            max_digits = digit_count
    return max_digits


# Returns the length, in number of digits, of value
def radix_get_length(value):
    if value == 0:
        return 1
```


Run

3.14 Overview of fast sorting algorithms

Fast sorting algorithm

A ***fast sorting algorithm*** is a sorting algorithm that has an average runtime complexity of O( ) or better. The table below shows average runtime complexities for several sorting algorithms.

Table 3.14.1: Sorting algorithms' average runtime complexity.

Sorting algorithm	Average case runtime complexity	Fast?
Selection sort	O( )	No
Insertion sort	O( )	No
Shell sort	O( )	No
Quicksort	O( )	Yes

Merge sort	O( )	Yes
Heap sort	O( )	Yes
Radix sort	O( )	Yes

PARTICIPATION ACTIVITY 3.14.1: Fast sorting algorithms.

1) Insertion sort is a fast sorting algorithm.
   - True
   - False

2) Merge sort is a fast sorting algorithm.
   - True
   - False

3) Radix sort is a fast sorting algorithm.
   - True
   - False

Comparison sorting

A ***element comparison sorting algorithm*** is a sorting algorithm that operates on an array of elements that can be compared to each other. Ex: An array of strings can be sorted with a comparison sorting algorithm, since two strings can be compared to determine if the one string is less than, equal to, or greater than another string. Selection sort, insertion sort, shell sort, quicksort, merge sort, and heap sort are all comparison sorting algorithms. Radix sort, in contrast, subdivides each array element into integer digits and is not a comparison sorting algorithm.

Table 3.14.2: Identifying comparison sorting algorithms.

Sorting algorithm	Comparison?
Selection sort	Yes
Insertion sort	Yes

Shell sort	Yes
Quicksort	Yes
Merge sort	Yes
Heap sort	Yes
Radix sort	No

PARTICIPATION ACTIVITY 3.14.2: Comparison sorting algorithms.

1) Selection sort can be used to sort an array of strings.
   - True
   - False

2) The fastest average runtime complexity of a comparison sorting algorithm is O( ).
   - True
   - False

Best and worst case runtime complexity

A fast sorting algorithm's best or worst case runtime complexity may differ from the average runtime complexity. Ex: The best and average case runtime complexity for quicksort is O( ), but the worst case is O( ).

Table 3.14.3: Fast sorting algorithm's best, average, and worst case runtime complexity.

Sorting algorithm	Best case runtime complexity	Average case runtime complexity	Worst case runtime complexity
Quicksort	O( )	O( )	O( )
Merge sort	O( )	O( )	O( )

Heap sort	O( )	O( )	O( )
Radix sort	O( )	O( )	O( )

PARTICIPATION ACTIVITY 3.14.3: Runtime complexity.

1) A fast sorting algorithm's worst case runtime complexity must be O( ) or better.
   - True
   - False

2) Which fast sorting algorithm's worst case runtime complexity is worse than O( )?
   - Quicksort
   - Heap sort
   - Radix sort

3.15 Python: Sorting with different operators

Sorting with Python built-ins

Many object types in Python have a *natural* ordering, meaning that when two objects of that type are compared, one comes before the other in some obvious way or the objects are equal. Ex: 3 comes before 4 in ascending numerical order and 'act' comes before 'ask' in ascending alphabetical order. Natural ordering is used by the comparison operators: <, <=, >, >=, == and !=.

Python has a built-in ***sorted()*** function that takes one list argument, sorts the list's elements in ascending order using the less than (<) operator, and returns a new list with the sorted elements.

Python lists also have a ***sort()*** method to do an in-place sorting of list elements in ascending order, meaning the list is modified and another list is not returned.

Figure 3.15.1: Built-in sorting in Python.

```
# Using sorted() function
num_list = [3, 7, 2, 8, 12, 4, 9, 5]
sorted_num_list = sorted(num_list)
print('UNSORTED:', num_list)
print('SORTED:', sorted_num_list)
print()

# Using sort() method
fruit_list = ["grape", "banana", "apple", "strawberry", "blueberry"]
print('UNSORTED:', fruit_list)
fruit_list.sort()
print('SORTED:', fruit_list)
```

```
UNSORTED: [3, 7, 2, 8, 12, 4, 9, 5]
SORTED: [2, 3, 4, 5, 7, 8, 9, 12]

UNSORTED: ['grape', 'banana', 'apple', 'strawberry', 'blueberry']
SORTED: ['apple', 'banana', 'blueberry', 'grape', 'strawberry']
```


Sorting in descending order with sorted() and sort()

sorted() and sort() can both take an optional keyword argument, reverse, that when assigned with True sorts a list in descending order.

Figure 3.15.2: Reverse sorting in Python.

```
num_list = [3, 7, 2, 8, 12, 4, 9, 5]
sorted_num_list = sorted(num_list, reverse = True)
print('UNSORTED:', num_list)
print('REVERSE SORTED:', sorted_num_list)
print()

fruit_list = ["grape", "banana", "apple", "strawberry", "blueberry"]
print('UNSORTED:', fruit_list)
fruit_list.sort(reverse = True)
print('REVERSE SORTED:', fruit_list)
```

```
UNSORTED: [3, 7, 2, 8, 12, 4, 9, 5]
REVERSE SORTED: [12, 9, 8, 7, 5, 4, 3, 2]

UNSORTED: ['grape', 'banana', 'apple', 'strawberry', 'blueberry']
REVERSE SORTED: ['strawberry', 'grape', 'blueberry', 'banana', 'apple']
```


PARTICIPATION ACTIVITY 3.15.1: sorted() and sort().

Assume num_list is a list defined in Python.

1) `num_list.sort()` returns a list.

- True
- False

2) The sorted() function has an order parameter that can be assigned with "ascending" or "descending".

- True
- False

Using a key function

The sorted() function and sort() method can take an optional key argument, which is a function that determines how to sort the list. One common use of the key argument is to sort a list of strings without regard to case. Ex: The list ["banana", "Grape", "Apple"] by default sorts with uppercase letters coming before lowercase letters: ["Apple", "Grape", "banana"]. To sort as if all letters are lowercase, sorted() is called with argument `key = str.lower`. Note: The changed list item ("apple" instead of "Apple") is only used temporarily during the sorting algorithm; the resulting list still has the original elements.

The argument is `key = str.lower`, not `key = str.lower()`, with parentheses. The key variable is assigned with the actual str.lower function itself, not by *calling* the str.lower() function.

Figure 3.15.3: Using str.lower as a key argument to sort a list of strings regardless of case.

```
# Program to sort strings using str.lower
file_names = [ "Grades.xls", "email.txt", "helper.py", "Test.java" ]
case_sensitive = sorted(file_names)
case_insensitive = sorted(file_names, key = str.lower)
print('Normal sort:', case_sensitive)
print('Case insensitive sort:', case_insensitive)
```

```
Normal sort: ['Grades.xls', 'Test.java', 'email.txt', 'helper.py']
Case insensitive sort: ['email.txt', 'Grades.xls', 'helper.py', 'Test.java']
```


Custom key function

Another common use for the key argument is to sort a list by one of the elements' values if each element forms a tuple. Ex: List elements that contain both a student name and ID number: [("Robert", 135216), ("Amir", 612901), ("Jennifer", 194821)]. A key argument can be used to sort such a list by name or by ID number.

Custom key functions must be defined to have only one parameter, a single list element. The function uses that element variable to return whatever the sorting key should be. Note again that the key parameter is assigned with the custom key function's *name* only, parentheses are not used.

Figure 3.15.4: Using sorted() with a custom key.


```
def key_is_name(element):
    return element[0]

def key_is_id(element):
    return element[1]

class_list = [("Robert", 135216), ("Amir", 612901), ("Jennifer", 194821),
("Ravi", 631104)]
name_result = sorted(class_list, key = key_is_name)
id_result = sorted(class_list, key = key_is_id)
print('Sort by name:', name_result)
print('Sort by id:', id_result)
```

```
Sort by name: [('Amir', 612901), ('Jennifer', 194821), ('Ravi', 631104), ('Robert',
135216)]
Sort by id: [('Robert', 135216), ('Jennifer', 194821), ('Amir', 612901), ('Ravi',
631104)]
```

PARTICIPATION ACTIVITY 3.15.2: Custom key functions.

1) A custom key function takes a list as an argument and returns the smallest item in the list.

   - True
   - False

2) The following function can be used as a key function to do a case-insensitive sort.

```
def custom_key(word):
    return word.lower()
```

   - True
   - False

3) The sorted() function and the list's sort() method both take an optional argument for a key function.

- ○ True
- ○ False

The itemgetter() function

A special function, ***itemgetter()***, defined in Python's operator module, can be used to get a key from an element using an index instead of a custom key function. Ex: `key = key_is_name` can be replaced with `key = operator.itemgetter(0)`, because the element names are found at index 0. itemgetter() takes an integer value argument and returns a *function* defined like `key_is_name(element)`. Ex. `operator.itemgetter(5)` returns a function that takes a list element argument, and the function returns `element[5]`.

Figure 3.15.5: Using sorted() with operator.itemgetter().

```
import operator
class_list = [("Robert", 135216), ("Amir", 612901), ("Jennifer", 194821),
("Ravi", 631104)]
name_result = sorted(class_list, key = operator.itemgetter(0))
id_result = sorted(class_list, key = operator.itemgetter(1))
print('Sort by name:', name_result)
print('Sort by id:', id_result)
```

```
Sort by name: [('Amir', 612901), ('Jennifer', 194821), ('Ravi', 631104), ('Robert',
135216)]
Sort by id: [('Robert', 135216), ('Jennifer', 194821), ('Amir', 612901), ('Ravi',
631104)]
```

PARTICIPATION ACTIVITY 3.15.3: Using operator.itemgetter() as a sorting key function.

Enter the contents of the resulting sorted lists in the format: `a, b, c`. If the code is invalid enter `Invalid`.

1)
```
num_list = [ (10, 14), (18,
12) ]
result = sorted(num_list, key
= operator.itemgetter(1))
```

[ ]

Check Show answer

2)
```
num_list = [ (10, 14, 0), (2,
1, 40), (6, 10, 4) ]
result = sorted(num_list, key
= operator.itemgetter(3))
```

[ ]

Check Show answer

3)
```
word_list = [ 'two', 'four',
'six', 'eight' ]
result = sorted(word_list,
key = operator.itemgetter(2))
```

[ ]

Check Show answer

zyDE 3.15.1: Using operator.itemgetter().

The program below displays the statistics for a few NHL hockey teams. Each item in the teams list is a 4-tuple in the form (Name, Wins, Losses, Total Points). The program displa the teams sorted by the team names. See if you can change the sorted() function call to display the teams sorted:

- in increasing order by wins
- in increasing order by points
- in decreasing order by points

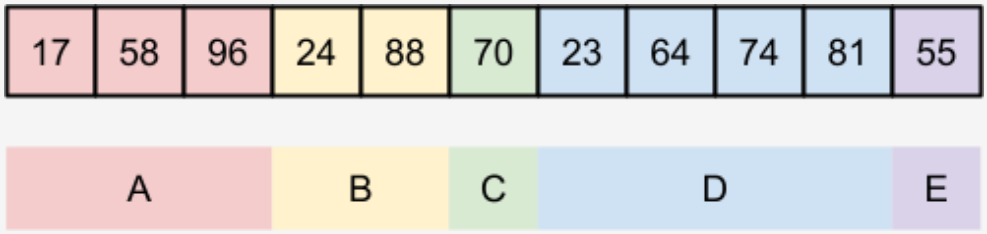
main.py Load default template... Run

```
1  import operator
2
3  teams = []
4  teams.append(('Lightning', 45, 17,
5  teams.append(('Capitals', 37, 21, 8
6  teams.append(('Predators', 42, 14,
7  teams.append(('Golden Knights', 41,
8
9  results = sorted(teams, key = opera
10 print('%15s%4s%4s%4s' % ("Team", "W
11
12 for team in results:
13     print('%15s%4d%4d%4d' % team)
14
```

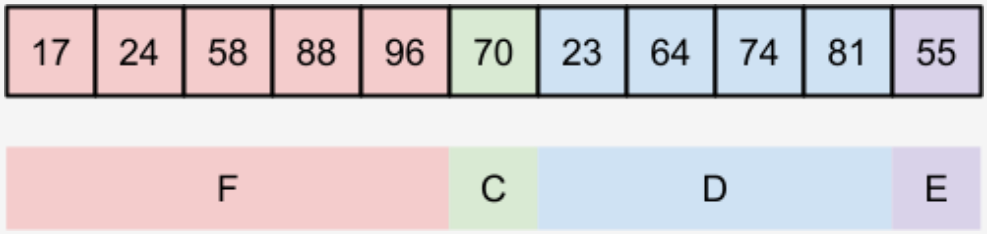
3.16 LAB: Natural merge sort

The merge sort algorithm recursively divides the list in half until a list with one element is reached. A variant of merge sort, called natural merge sort, instead finds already-sorted runs of elements and merges the runs together.

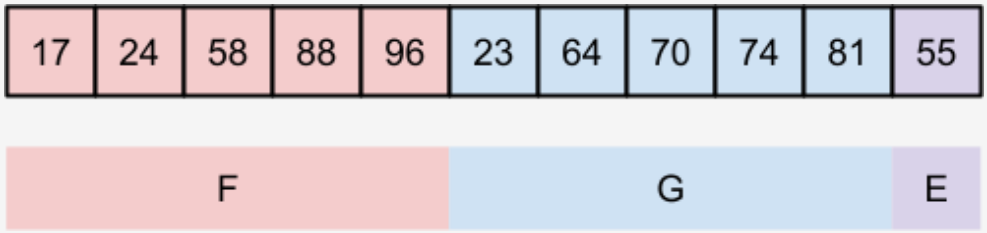
Ex: The unsorted list below has five sorted runs.



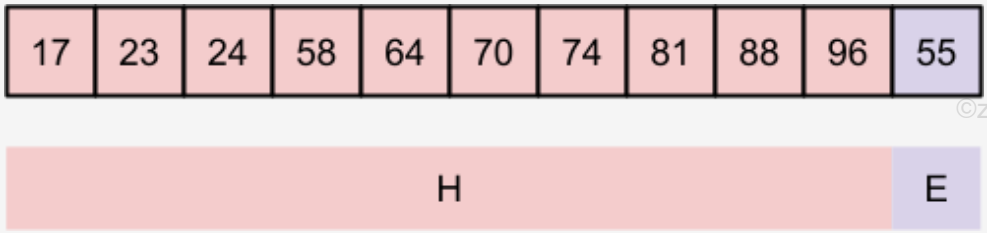
Natural merge sort starts at index 0 and finds sorted runs A and B. Runs A and B are merged, using the same merging algorithm as merge sort, to make run F.



Next, the algorithm starts after the merged part F, again looking for two sequential, sorted runs. Runs C and D are found and merged to make run G.



The algorithm then starts after the merged portion G. Only one run exists, run E, until the end of the list is reached. So the algorithm starts back at index 0, finds runs F and G, and merges to make run H.



Again a single run is found after the just-merged part, so the search starts back at index 0. Runs H and E are found and merged.

One last search for a sorted run occurs, finding a sorted run length equal to the list's length. So the list is sorted and the algorithm terminates.

17	23	24	55	58	64	70	74	81	88	96

Step 1: Implement the get_sorted_run_length() method

Implement the get_sorted_run_length() method in NaturalMergeSorter.py. Access NaturalMergeSorter.py by clicking on the orange arrow next to main.py at the top of the coding window.

get_sorted_run_length() has two parameters:

- **integer_list**: a list of integers and
- **start_index**: an integer for the run's starting index.

The method returns the number of list elements sorted in ascending order, starting at start_index and ending either at the end of the sorted run, or the end of the list, whichever comes first. The method returns 0 if start_index is out of bounds.

File main.py has several test cases for get_sorted_run_length() that can be run by clicking the "Run program" button. One test case also exists for natural_merge_sort(), but that can be ignored until step two is completed.

The program's output does not affect grading.

Submit for grading to ensure that the get_sorted_run_length() unit tests pass before proceeding.

Step 2: Implement the natural_merge_sort() method

Implement the natural_merge_sort() method in NaturalMergeSorter.py. natural_merge_sort() must:

1. Start at index `i`=0
2. Get the length of the first sorted run, starting at `i`
    - Return if the first run's length equals the list's length
    - If the first run ends at the list's end, reassign `i`=0 and repeat step 2
3. Get the length of the second sorted run, starting immediately after the first
4. Merge the two runs with the provided merge() method
5. Reassign `i` with the first index after the second run, or 0 if the second run ends at the list's end
6. Go to step 2

LAB ACTIVITY | 3.16.1: LAB: Natural merge sort | 10 / 10

Downloadable files

`main.py` , `RunLengthTestCase.py` , and `NaturalMergeSorter.py`

Download

File is marked as read only Current file: main.py


```
1 Loading latest submission...
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

Retrieving signature