10.1 Sorting: Introduction

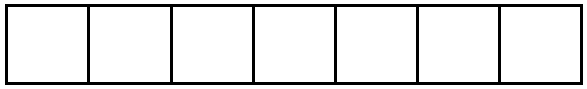
Sorting is the process of converting a list of elements into ascending (or descending) order. For example, given a list of numbers (17, 3, 44, 6, 9), the list after sorting is (3, 6, 9, 17, 44). You may have carried out sorting when arranging papers in alphabetical order, or arranging envelopes to have 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 10.1.1: 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 10.1.2: 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

10.2 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 10.2.1: 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 10.2.2: 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

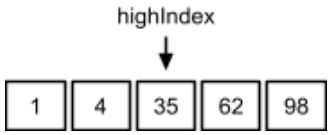
```
pivot =
```

Check Show answer

PARTICIPATION ACTIVITY 10.2.3: Low and high partitions.

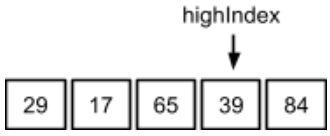
Determine if the low and high partitions are correct given highIndex and pivot.

1) pivot = 35



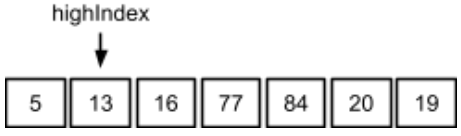
- Correct
- Incorrect

2) pivot = 65



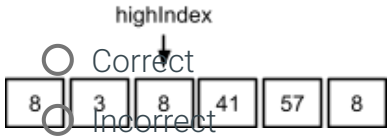
- Correct
- Incorrect

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 10.2.4: 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 10.2.1: Quicksort algorithm.

©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
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
```

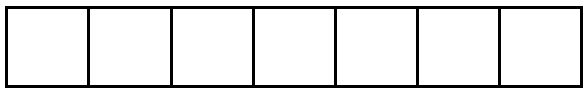
Quicksort activity

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 10.2.5: 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 10.2.6: 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 10.2.7: 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 10.2.1: Quicksort.

489394.3384924.qx3zqy7

Start

Given numbers = (16, 28, 39, 57, 95, 54, 45), lowIndex = 0, highIndex = 6

What is the midpoint? Ex: 9

What is the pivot?

1	2	3	4	5	6

Check Next

10.3 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 | 10.3.1: 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 | 10.3.2: 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 10.3.3: 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:

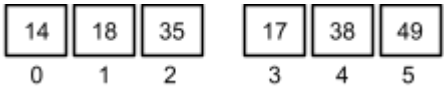
undefined

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 10.3.4: 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 10.3.1: Merge sort algorithm.

```
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 10.3.5: 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 10.3.1: Merge sort.

489394.3384924.qx3zqy7

Start

numbers:

66	62	42	58	96	65	41	76

What call sorts the numbers array?

MergeSort(numbers, Ex: 1 , ____)

1 2 3 4 5

Check Next

10.4 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 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( ). Bubble sort is often considered impractical for real-world use because many faster sorting algorithms exist.

Figure 10.4.1: 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 10.4.1: 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( ).
   - True
   - False

10.5 Selection sort

This section has been set as optional by your instructor.

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 10.5.1: 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 10.5.2: 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 10.5.1: 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()
}
```


©zyBooks 06/21/23 22:37 1692462
Taylor Larrechea
COLORADOCSPB2270Summer2023

```
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 10.5.3: 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 10.5.1: Selection sort.

489394.3384924.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

10.6 Insertion sort

This section has been set as optional by your instructor.

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 10.6.1: 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 10.6.1: 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 10.6.2: 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 10.6.3: 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 10.6.4: 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 10.6.5: 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 10.6.6: 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 10.6.1: Insertion sort.

489394.3384924.qx3zqy7

Start

Given list (20, 25, 26, 35, 37, 39, 24, 38, 30), what is the value of i when the first swap executes

Ex: 1

1 2 3 4 5

Check Next

10.7 Shell sort

This section has been set as optional by your instructor.

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 10.7.1: 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 10.7.2: 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
   - ○ 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 10.7.3: 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 10.7.4: 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 10.7.5: 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

10.7.6: 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 10.7.1: Shell sort.

489394.3384924.qx3zqy7

Start

Given an array [80, 88, 40, 55, 94, 24, 74, 54, 81] and a gap value of 4:

What is the first interleaved array?

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

What is the second interleaved array?

[ ]

1 2 3

Check Next

10.8 Radix sort

This section has been set as optional by your instructor.

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 10.8.1: 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 10.8.2: Using the 1's digit, place each integer in the correct bucket.

If unable to drag and drop, refresh the page.

49 74 7 50

Bucket 0

	Bucket 4
	Bucket 7
	Bucket 9

Reset

PARTICIPATION ACTIVITY 10.8.3: Using the 10's digit, place each integer in the correct bucket.

If unable to drag and drop, refresh the page.

7 86 50 74

	Bucket 0
	Bucket 5
	Bucket 7
	Bucket 8

Reset

PARTICIPATION ACTIVITY 10.8.4: 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 10.8.5: 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 10.8.6: 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 10.8.1: RadixGetMaxLength and RadixGetLength functions.

```
// 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 10.8.7: 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 10.8.8: 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 10.8.2: 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 10.8.9: 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 10.8.1: Radix sort.

489394.3384924.qx3zqy7

Start

Using only the 1's digit,
what is the correct bucket for 96? Ex: 5
what is the correct bucket for 408?

Using only the 10's digit,
what is the correct bucket for 96?
what is the correct bucket for 408?

Using only the 100's digit,
what is the correct bucket for 96?
what is the correct bucket for 408?

	Buckets using 1's digit	Buckets using 10's digit	Buckets using 100's
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			

1 2 3 4

Check Next

10.9 Overview of fast sorting algorithms

This section has been set as optional by your instructor.

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 10.9.1: 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 10.9.1: 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 10.9.2: 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 10.9.2: 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 10.9.3: 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 10.9.3: 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