8.1 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 8.1.1: 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 | 8.1.2: 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^{th} 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 8.1.1: Selection sort algorithm.

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

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

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

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

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

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

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

   SelectionSort(numbers, NUMBERS_SIZE)

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



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


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

PARTICIPATION ACTIVITY 8.1.3: Selection sort runtime.

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

Check Show answer

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

Check Show answer

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

Check Show answer

CHALLENGE ACTIVITY 8.1.1: Selection sort.

Start

When using selection sort to sort a list with \(14\) elements, what is the minimum number of assignments to \(\texttt{indexSmallest}\)?

1	2	3	4

Check Next

8.2 Quickselect

Quickselect is an algorithm that selects the k^{th} smallest element in a list. Ex: Running quickselect on the list (15, 73, 5, 88, 9) with k = 0, returns the smallest element in the list, or 5.

For a list with N elements, quickselect uses quicksort's partition function to partition the list into a low partition containing the X smallest elements and a high partition containing the N-X largest elements. The k^{th} smallest element is in the low partition if k is ≤ the last index in the low partition, and in the high partition otherwise. Quickselect is recursively called on the partition that contains the k^{th} element. When a partition of size 1 is encountered, quickselect has found the k^{th} smallest element.

Quickselect partially sorts the list when selecting the k^{th} smallest element.

The best case and average runtime complexity of quickselect are both O(N). In the worst case, quickselect may sort the entire list, resulting in a runtime of O(N^2).

Figure 8.2.1: Quickselect algorithm.

```
// Selects kth smallest element, where k is 0-based
Quickselect(numbers, first, last, k) {
   if (first >= last)
      return numbers[first]

   lowLastIndex = Partition(numbers, first, last)

   if (k <= lowLastIndex)
      return Quickselect(numbers, first, lowLastIndex, k)
   return Quickselect(numbers, lowLastIndex + 1, last, k)
}
```


PARTICIPATION ACTIVITY 8.2.1: Quickselect.

1) Calling quickselect with argument k equal to 1 returns the smallest element in the list.

- True
- False

2) The following function produces the same result as quickselect, albeit with a different runtime complexity.

```
Quickselect(numbers, first,
last, k) {
   Quicksort(numbers, first,
last)
   return numbers[k]
}
```

- True
- False

3) Given k = 4, if the quickselect call `Partition(numbers, 0, 10)` returns 4, then the element being selected is in the low partition.

- True
- False

8.3 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 8.3.1: 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 8.3.1: 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 | 8.3.2: Insertion sort algorithm execution.

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

1) Given list (20, 14, 85, 3, 9), what value will be in the 0^{th} element after the first pass over the outer loop (i = 1)?

Check Show answer

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

Check Show answer

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

Check Show answer

Insertion sort runtime

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

PARTICIPATION ACTIVITY | 8.3.3: 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 | 8.3.4: 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 | 8.3.5: 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 | 8.3.6: 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 8.3.1: Insertion sort.

Start

Given list (26, 28, 29, 33, 39, 30, 37), what is the value of i when the first swap execu

Ex: 1

1	2	3	4	5

Check Next

8.4 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 | 8.4.1: 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 | 8.4.2: 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 | 8.4.3: 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 | 8.4.4: Insertion sort variant.

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

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

2) Given the call InsertionSortInterleaved(values, 4, 1, 4), what is the initial value of the loop variable i?

- 0
- 1
- 4
- 5

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

- True
- False

4) If a gap value of 2 is chosen, then the following 2 function calls will fully sort a list:
InsertionSortInterleaved(list, 9, 0, 2)
InsertionSortInterleaved(list, 9, 1, 2)

- True
- False

Shell sort algorithm

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

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

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

PARTICIPATION ACTIVITY | 8.4.5: 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 | 8.4.6: 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

8.5 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 | 8.5.1: 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 | 8.5.2: 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 | 8.5.3: 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 | 8.5.4: Tracing merge operation.

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

14	18	35	17	38	49
0	1	2	3	4	5

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 8.5.1: 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)
```

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 8.5.5: 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 8.5.1: Merge sort.

Start

numbers:

77	33	58	99	79	83	55	62	94	32

What call sorts the numbers array?

MergeSort(numbers, Ex: 1 ,)

1 2 3 4 5

Check Next

8.6 Heap sort

Heapify operation

Heapsort is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order. An array of unsorted values must first be converted into a heap. The ***heapify*** operation is used to turn an array into a heap. Since leaf nodes already satisfy the max heap property, heapifying to build a max-heap is achieved by percolating down on every non-leaf node in reverse order.

PARTICIPATION ACTIVITY 8.6.1: Heapify operation.

Animation captions:

1. If the original array is represented in tree form, the tree is not a valid max-heap.
2. Leaf nodes always satisfy the max heap property, since no child nodes exist that can contain larger keys. Heapification will start on node 92.
3. 92 is greater than 24 and 42, so percolating 92 down ends immediately.
4. Percolating 55 down results in a swap with 98.
5. Percolating 77 down involves a swap with 98. The resulting array is a valid max-heap.

The heapify operation starts on the internal node with the largest index and continues down to, and including, the root node at index 0. Given a binary tree with N nodes, the largest internal node index is $\lfloor N/2 \rfloor$ - 1.

Table 8.6.1: Max-heap largest internal node index.

Number of nodes in binary heap	Largest internal node index
1	-1 (no internal nodes)
2	0
3	0
4	1
5	1
6	2
7	2
...	...
N	$\lfloor N/2 \rfloor$ - 1

PARTICIPATION ACTIVITY 8.6.2: Heapify operation.

1) For an array with 7 nodes, how many percolate-down operations are necessary to heapify the array?

Check Show answer

2) For an array with 10 nodes, how many percolate-down operations are necessary to heapify the array?

Check Show answer

PARTICIPATION ACTIVITY | 8.6.3: Heapify operation - critical thinking.

1) An array sorted in ascending order is already a valid max-heap.

- True
- False

2) Which array could be heapified with the fewest number of operations, including all swaps used for percolating?

- (10, 20, 30, 40)
- (30, 20, 40, 10)
- (10, 10, 10, 10)

Heapsort overview

Heapsort begins by heapifying the array into a max-heap and initializing an end index value to the size of the array minus 1. Heapsort repeatedly removes the maximum value, stores that value at the end index, and decrements the end index. The removal loop repeats until the end index is 0.

PARTICIPATION ACTIVITY | 8.6.4: Heapsort.

Animation captions:

1. The array is heapified first. Each internal node is percolated down, from highest node index to lowest.
2. The end index is initialized to 6, to refer to the last item. 94's "removal" starts by swapping with 68.
3. Removing from a heap means that the rightmost node on the lowest level

disappears before the percolate down. End index is decremented after percolating.
4. 88 is swapped with 49, the last node disappears, and 49 is percolated down.
5. The process continues until end index is 0.
6. The array is sorted.

PARTICIPATION ACTIVITY | 8.6.5: Heapsort.

Suppose the original array to be heapified is (11, 21, 12, 13, 19, 15).

1) The percolate down operation must be performed on which nodes?
- 15, 19, and 13
- 12, 21, and 11
- All nodes in the heap

2) What are the first 2 elements swapped?
- 11 and 21
- 21 and 13
- 12 and 15

3) What are the last 2 elements swapped?
- 11 and 19
- 11 and 21
- 19 and 21

4) What is the heapified array?

- (11, 21, 12, 13, 19, 15)
- (21, 19, 15, 13, 12, 11)
- (21, 19, 15, 13, 11, 12)

Heapsort algorithm

Heapsort uses 2 loops to sort an array. The first loop heapifies the array using MaxHeapPercolateDown. The second loop removes the maximum value, stores that value at the end index, and decrements the end index, until the end index is 0.

Figure 8.6.1: Heap sort.


```
Heapsort(numbers, numbersSize) {
   // Heapify numbers array
   for (i = numbersSize / 2 - 1; i >= 0; i--) {
      MaxHeapPercolateDown(i, numbers, numbersSize)
   }

   for (i = numbersSize - 1; i > 0; i--) {
      Swap numbers[0] and numbers[i]
      MaxHeapPercolateDown(0, numbers, i)
   }
}
```

PARTICIPATION ACTIVITY 8.6.6: Heapsort algorithm.

1) How many times will MaxHeapPercolateDown be called by Heapsort when sorting an array with 10 elements?

- 5
- 10
- 14
- 20

2) Calling Heapsort on an array with 1 element will cause an out of bounds array access.

- True
- False

3) Heapsort's worst-case runtime is O(N log N).

True

False

4) Heapsort uses recursion.

True

False

CHALLENGE ACTIVITY | 8.6.1: Heap sort.

8.7 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 | 8.7.1: A particular single digit in an integer can determine the integer's bucket.

Animation captions:

1. Using only the 1's digit, each value 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

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

50 **7** **74** **49**

Bucket 0

Bucket 4

Bucket 7

Bucket 9

Reset

PARTICIPATION ACTIVITY

8.7.3: Using the 10's digit, place each integer in the correct bucket.

86 **50** **74** **7**

Bucket 0

Bucket 5

Bucket 7

Bucket 8

Reset

PARTICIPATION ACTIVITY

8.7.4: 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 | 8.7.5: 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 | 8.7.6: 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 8.7.1: 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 8.7.7: 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 | 8.7.8: 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

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 algorithms 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 8.7.2: 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 = GetLowestDigit(array[i] / pow10)
         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 8.7.9: 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 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.

8.8 Sorting linked lists

Insertion sort for doubly-linked lists

Insertion sort for a doubly-linked list operates similarly to the insertion sort algorithm used for arrays. Starting with the second list element, each element in the linked list is visited. Each visited element is moved back as needed and inserted into the correct position in the list's sorted portion. The list must be a doubly-linked list, since backward traversal is not possible in a singly-linked list.

PARTICIPATION ACTIVITY 8.8.1: Sorting a doubly-linked list with insertion sort.

Animation content:

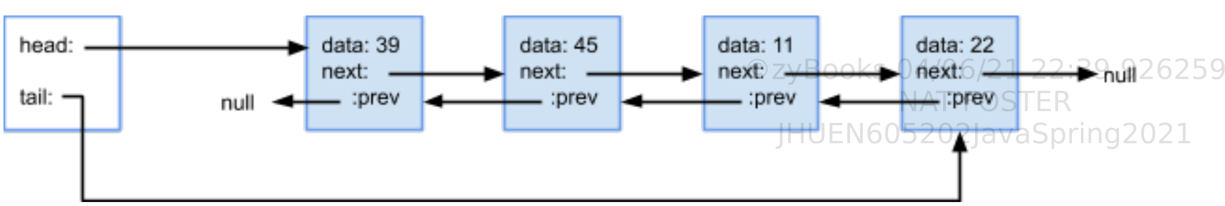
`undefined`

Animation captions:

1. The curNode pointer begins at node 91 in the list.
2. searchNode starts at node 81 and does not move because 81 is not greater than 91. Removing and re-inserting node 91 after node 81 does not change the list.
3. For node 23, searchNode traverses the list backward until becoming null. Node 23 is prepended as the new list head.
4. Node 49 is inserted after node 23, using ListInsertAfter.
5. Node 12 is inserted before node 23, using ListPrepend, to complete the sort.

PARTICIPATION ACTIVITY 8.8.2: Insertion sort for doubly-linked lists.

Suppose ListInsertionSortDoublyLinked is executed to sort the list below.



1) What is the first node that curNode will point to?

- Node 39
- Node 45
- Node 11

2) The ordering of list nodes is not altered when node 45 is removed and then inserted after node 39.

- True
- False

3) ListPrepend is called on which node(s)?

- Node 11 only
- Node 22 only
- Nodes 11 and 22

Algorithm efficiency

Insertion sort's typical runtime is O(N^2). If a list has N elements, the outer loop executes N - 1 times. For each outer loop execution, the inner loop may need to examine all elements in the sorted part. Thus, the inner loop executes on average $N/2$ times. So the total number of comparisons is proportional to $(N-1) \cdot (N/2)$, or O(N^2). In the best case scenario, the list is already sorted, and the runtime complexity is O(N).

Insertion sort for singly-linked lists

Insertion sort can sort a singly-linked list by changing how each visited element is inserted into the sorted portion of the list. The standard insertion sort algorithm traverses the list from the current element toward the list head to find the insertion position. For a singly-linked list, the insertion sort algorithm can find the insertion position by traversing the list from the list head toward the current element.

Since a singly-linked list only supports inserting a node after an existing list node, the ListFindInsertionPosition algorithm searches the list for the insertion position and returns the list node after which the current node should be inserted. If the current node should be inserted at the head, ListFindInsertionPosition return null.

PARTICIPATION ACTIVITY | 8.8.3: Sorting a singly-linked list with insertion sort.

Animation content:

undefined

Animation captions:

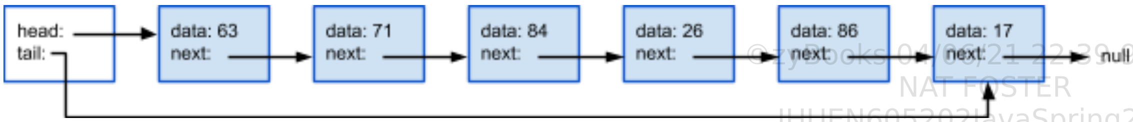
1. Insertion sort for a singly-linked list initializes curNode to point to the second list element, or node 56.
2. ListFindInsertionPosition searches the list from the head toward the current node to find the insertion position. ListFindInsertionPosition returns null, so Node 56 is prepended as the list head.
3. Node 64 is less than node 71. ListFindInsertionPosition returns a pointer to the node before node 71, or node 56. Then, node 64 is inserted after node 56.
4. The insertion position for node 87 is after node 71. Node 87 is already in the correct position and is not moved.
5. Although node 74 is only moved back one position, ListFindInsertionPosition compared node 74 with all other nodes' values to find the insertion position.

Figure 8.8.1: ListFindInsertionPosition algorithm.

```
ListFindInsertionPosition(list, dataValue) {
   curNodeA = null
   curNodeB = list⇢head
   while (curNodeB != null and dataValue > curNodeB⇢data) {
      curNodeA = curNodeB
      curNodeB = curNodeB⇢next
   }
   return curNodeA
}
```

PARTICIPATION ACTIVITY | 8.8.4: Sorting singly-linked lists with insertion sort.

Given ListInsertionSortSinglyLinked is called to sort the list below.



1) What is returned by the first call to ListFindInsertionPosition?

- null
- Node 63
- Node 71
- Node 84

2) How many times is ListPrepend called?

- 0
- 1
- 2

3) How many times is ListInsertAfter called?

- 0
- 1
- 2

Algorithm efficiency

The average and worst case runtime of ListInsertionSortSinglyLinked is O(N^2). The best case runtime is O(N), which occurs when the list is sorted in descending order.

Sorting linked-lists vs. arrays

Sorting algorithms for arrays, such as quicksort and heapsort, require constant-time access to arbitrary, indexed locations to operate efficiently. Linked lists do not allow indexed access, making for difficult adaptation of such sorting algorithms to operate on linked lists. The tables below provides a brief overview of the challenges in adapting array sorting algorithm for linked lists.

Table 8.8.1: Sorting algorithms easily adapted to efficiently sort linked lists.

Sorting algorithm	Adaptation to linked lists
Insertion sort	Operates similarly on doubly-linked lists. Requires searching from the head of the list for an element's insertion position for singly-linked lists.
Merge sort	Finding the middle of the list requires searching linearly from the head of the list. The merge algorithm can also merge lists without additional storage.

Table 8.8.2: Sorting algorithms that cannot as efficiently sort linked lists.

Sorting algorithm	Challenge
Shell sort	Jumping the gap between elements cannot be done on a linked list, as each element between two elements must be traversed.
Quicksort	Partitioning requires backward traversal through the right portion of the array. Singly-linked lists do not support backward traversal.
Heap sort	Indexed access is required to find child nodes in constant time when percolating down.

PARTICIPATION ACTIVITY 8.8.5: Sorting linked-lists vs. sorting arrays.

1) What aspect of linked lists makes adapting array-based sorting algorithms to linked lists difficult?

- Two elements in a linked list cannot be swapped in constant time.
- Nodes in a linked list cannot be moved.
- Elements in a linked list cannot be accessed by index.

2) Which sorting algorithm uses a gap value to jump between elements, and is difficult to adapt to linked lists for this reason?

- Insertion sort
- Merge sort
- Shell sort

3) Why are sorting algorithms for arrays generally more difficult to adapt to singly-linked lists than to doubly-linked lists?

- Singly-linked lists do not support backward traversal.
- Singly-linked do not support inserting nodes at arbitrary locations.

8.9 Topological sort

This section has been set as optional by your instructor.

Overview

A ***topological sort*** of a directed, acyclic graph produces a list of the graph's vertices such that for every edge from a vertex X to a vertex Y, X comes before Y in the list.

PARTICIPATION ACTIVITY 8.9.1: Topological sort.

Animation captions:

1. Analysis of each edge in the graph determines if an ordering of vertices is a valid topological sort.
2. If an edge from X to Y exists, X must appear before Y in a valid topological sort. C, D, A, F, B, E is not valid because this requirements is violated for three edges.
3. Ordering D, A, F, E, C, B has 1 edge violating the requirement, so the ordering is not a valid topological sort.

4. For ordering D, A, F, E, B, C, the requirement holds for all edges, so the ordering is a valid topological sort.
5. A graph can have more than 1 valid topological sort. Another valid ordering is D, A, F, B, E, C.

PARTICIPATION ACTIVITY | 8.9.2: Topological sort.

1) In the example above, D, A, B, F, E, C is a valid topological sort.

- True
- False

2) Which of the following is NOT a requirement of the graph for topological sorting?

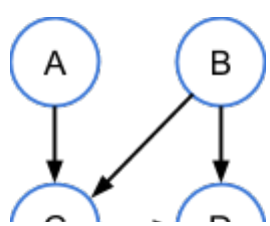
- The graph must be acyclic.
- The graph must be directed.
- The graph must be weighted.

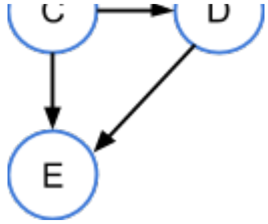
3) For a directed, acyclic graph, only one possible topological sort output exists.

- True
- False

PARTICIPATION ACTIVITY | 8.9.3: Identifying valid topological sorts.

Indicate whether each vertex ordering is a valid topological sort of the graph below.





1) A, B, C, D, E

- Valid
- Invalid

2) E, D, C, B, A

- Valid
- Invalid

3) D, E, A, B, C

- Valid
- Invalid

4) B, A, C, D, E

- Valid
- Invalid

5) B, A, C, E, D

- Valid
- Invalid

Example: course prerequisites

Graphs can be used to indicate a sequence of steps, where an edge from X to Y indicates that X must be done before Y. A topological sort of such a graph provides one possible ordering for performing the steps. Ex: Given a graph representing course prerequisites, a topological sort of the graph provides an ordering in which the courses can be taken.

PARTICIPATION ACTIVITY 8.9.4: Topological sorting can be used to order course prerequisites.

Animation captions:

1. For a graph representing course prerequisites, the vertices represent courses, and the edges represent the prerequisites. CS 101 must be taken before CS 102, and CS 102 before CS 103.
2. CS 103 is "Robotics Programming" and has a physic course (Phys 101) as a prerequisite. Phys 101 also has a math prerequisite (Math 101).
3. The graph's valid topological sorts provide possible orders in which to take the courses.

PARTICIPATION ACTIVITY | 8.9.5: Course prerequisites.

1) The "Math 101" and "CS 101" vertices have no incoming edges, and therefore one of these two vertices must be the first vertex in any topological sort.

- True
- False

2) Every topological sort ends with the "CS 103" vertex because this vertex has no outgoing edges.

- True
- False

Topological sort algorithm

The topological sort algorithm uses three lists: a results list that will contain a topological sort of vertices, a no-incoming-edges list of vertices with no incoming edges, and a remaining-edges list. The result list starts as an empty list of vertices. The no-incoming-edges vertex list starts as a list of all vertices in the graph with no incoming edges. The remaining-edges list starts as a list of all edges in the graph.

The algorithm executes while the no-incoming-edges vertex list is not empty. For each iteration, a vertex is removed from the no-incoming-edges list and added to the result list.

Next, a temporary list is built by removing all edges in the remaining-edges list that are outgoing from the removed vertex. For each edge currentE in the temporary list, the number of edges in the remaining-edges list that are incoming to currentE's terminating vertex are counted. If the incoming edge count is 0, then currentE's terminating vertex is added to the no-incoming-edges vertex list.

Because each loop iteration can remove any vertex from the no-incoming-edges list, the algorithm's output is not guaranteed to be the graph's only possible topological sort.

PARTICIPATION ACTIVITY | 8.9.6: Topological sort algorithm.

Animation content:

undefined

Animation captions:

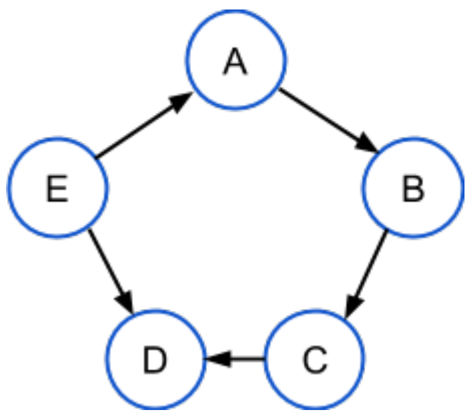
1. The topological sort algorithm begins by initializing an empty result list, a list of all vertices with no incoming edges, and a "remaining edges" list with all edges in the graph.
2. Vertex E is removed from the list of vertices with no incoming edges and added to resultList. Outgoing edges from E are removed from remainingEdges and added to outgoingEdges.
3. Edge EF goes to vertex F, which still has 2 incoming edges. Edge EG goes to vertex G, which still has 1 incoming edge.
4. Vertex A is removed and added to resultList. Outgoing edges from A are removed from remainingEdges. Vertices B and C are added to noIncoming.
5. Vertices C and B are processed, each with 1 outgoing edge.
6. Vertices B and F are processed, each also with 1 outgoing edge.
7. Vertex G is processed last. No outgoing edges remain. The final result is E, A, C, D, B, F, G.

Figure 8.9.1: GraphGetIncomingEdgeCount function.

```
GraphGetIncomingEdgeCount(edgeList, vertex) {
   count = 0
   for each edge currentE in edgeList {
      if (edge⇢toVertex == vertex)
         count = count + 1
   }
   return count
}
```


PARTICIPATION ACTIVITY 8.9.7: Topological sort algorithm.

Consider calling GraphTopologicalSort on the graph below.



1) In the first iteration of the while loop, what is assigned to currentV?

- Vertex A
- Vertex E
- Undefined

2) In the first iteration of the while loop, what is the contents of outgoingEdges right before the for-each loop begins?

- Edge from E to A and edge from E to D
- Edge from A to B
- No edges

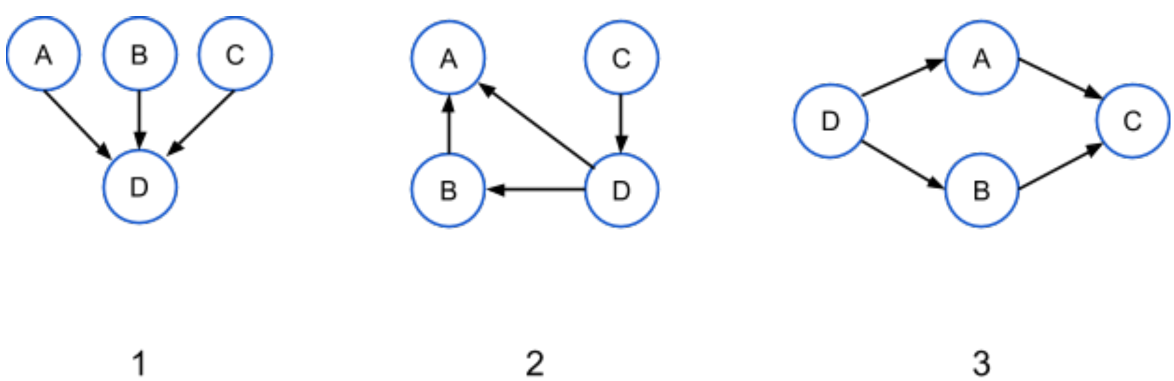
3) When currentV becomes vertex C, what is the contents of resultList?

- A, B
- E, A, B
- E, D

4) What is the final contents of resultList?

- A, B, C, D, E
- E, A, B, C, D
- E, A, B, D, C

PARTICIPATION ACTIVITY 8.9.8: Topological sort matching.



Graph 3 **Graph 2** **Graph 1**

D, B, A, C

C, D, B, A

B, C, A, D

Reset

PARTICIPATION ACTIVITY | 8.9.9: Topological sort algorithm.

1) What does `GraphTopologicalSort` return?

- ○ A list of vertices.
- ○ A list of edges.
- ○ A list of indices.

2) GraphTopologicalSort will not work on a graph with a positive number of vertices but no edges.

- ○ True
- ○ False

3) If a graph implementation stores incoming and outgoing edge counts in each vertex, then the statement `GraphGetIncomingEdgeCount(remainingEdges, edge⇢to)` can be replaced with `currentE⇢toVertex⇢incomingEdgeCount`.

- ○ True
- ○ False

Algorithm efficiency

The two vertex lists used in the topological sort algorithm will at most contain all the vertices in the graph. The remaining-edge list will at most contain all edges in the graph. Therefore, for a graph with a set of vertices V and a set of edges E, the space complexity of topological sorting is $O(|V| + |E|)$. If a graph implementation allows for retrieval of a vertex's incoming and outgoing edges in constant time, then the time complexity of topological sorting is also $O(|V| + |E|)$.

8.10 Overview of fast sorting algorithms

Fast sorting algorithm

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

Table 8.10.1: Sorting algorithms' average runtime complexity.

Sorting algorithm	Average case runtime complexity	Fast?
Selection sort	O(N^2)	No
Insertion sort	O(N^2)	No
Shell sort	O($N^{1.5}$)	No
Quicksort	O($NlogN$)	Yes
Merge sort	O($NlogN$)	Yes
Heap sort	O($NlogN$)	Yes
Radix sort	O(N)	Yes

PARTICIPATION ACTIVITY 8.10.1: 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

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 8.10.2: 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 8.10.2: 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($NlogN$).

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($NlogN$), but the worst case is O(N^2).

Table 8.10.3: 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($NlogN$)	O($NlogN$)	O(N^2)
Merge sort	O($NlogN$)	O($NlogN$)	O($NlogN$)
Heap sort	O(N)	O($NlogN$)	O($NlogN$)
Radix sort	O(N)	O(N)	O(N)

PARTICIPATION ACTIVITY 8.10.3: Runtime complexity.

1) A fast sorting algorithm's worst case runtime complexity must be O($NlogN$) or better.

True

False

2) Which fast sorting algorithm's worst case runtime complexity is worse than O($NlogN$)?

- Quicksort
- Heap sort
- Radix sort

8.11 Huffman compression

Basic compression idea

Given data represented as some quantity of bits, ***compression*** transforms the data to use fewer bits. Compressed data uses less storage and can be communicated faster than uncompressed data.

The basic idea of compression is to encode frequently-occurring items using fewer bits. Ex: Uncompressed ASCII characters use 8 bits each, but compression uses fewer than 8 bits for more frequently occurring characters.

PARTICIPATION ACTIVITY 8.11.1: The basic idea of compression is to use fewer bits for frequent items (and more bits for less-frequent items).

Animation captions:

1. The text "AAA Go" as ASCII would use 6 * 8 = 48 bits. Such data is uncompressed.
2. Compression uses a dictionary of codes specific to the data. Frequent items get shorter codes. Here, A (which is most frequent) is 0, space 10, G 110, and o 111.
3. Thus, "AAA Go" is compressed as 0 0 0 10 110 111. The compressed data uses only eleven bits, much fewer than the 48 bits uncompressed.

PARTICIPATION ACTIVITY 8.11.2: Basic compression.

Given the following dictionary:

00000000: 00
11111111: 01
00000010: 10
00000011: 110
00000100: 111

1) Compress the following:
00000000 00000000 11111111
00000100

Check Show answer

2) Compress the following:
00000011 00000010

Check Show answer

3) Decompress the following: 00 01 00

Check Show answer

4) Is any code in the dictionary a prefix of another code? Type yes or no.

Check Show answer

5) Decompress the following, in which the spaces that were inserted above for reading convenience are absent: 0011000.

Building a character frequency table

Prior to compression, a character frequency table must be built for an input string. Such a table contains each distinct character from the input string and each character's number of occurrences.

Programming languages commonly provide a dictionary or map object to store the character frequency table.

PARTICIPATION ACTIVITY 8.11.3: Building a character frequency table.

Animation content:

Animation captions:

1. A new dictionary is created for the character frequency table and iteration through the input string's characters begins.
2. The current character, A, is not in the dictionary. So A is added to the dictionary with a frequency of 1.
3. The next character, P, is also not in the dictionary and is added with a frequency of 1.
4. The next character, P, is already in the table, so the existing frequency is incremented from 1 to 2.
5. For remaining characters, first occurrences set the frequency to 1 and subsequent occurrences increment the existing frequency.

PARTICIPATION ACTIVITY 8.11.4: Character frequency counts.

Given the text "seems he fleed", indicate the frequency counts.

1) s

2) e

- 1
- 2
- 2
- 3
- 5
- 6

3) Each m, h, f, l, and d

- 1
- 2
- 3

4) (space)

- 1
- 2
- 3

Huffman coding

Huffman coding is a common compression technique that assigns fewer bits to frequent items, using a binary tree.

PARTICIPATION ACTIVITY | 8.11.5: A binary tree can be used to determine the Huffman coding.

Animation captions:

1. Huffman coding first determines the frequencies of each item. Here, a occurs 4 times, b 3, c 2, and d 1. (Total is 10).
2. Each item is a "leaf node" in a tree. The pair of nodes yielding the lowest sum is found, and merged into a new node formed with that sum. Here, c and d yield 2 + 1 = 3.
3. The merging continues. The lowest sum is b's 3 plus the new node's 3, yielding 6. (Note that c and d are no longer eligible nodes). The merging ends when only 1 node exists.
4. Each leaf node's encoding is obtained by traversing from the top node to the left. Each left branch appends a 0, and each right branch appends a 1, to the code.

PARTICIPATION ACTIVITY 8.11.6: Huffman coding example: Merging nodes.

A 100-character text has these character frequencies:
A: 50
C: 40
B: 4
D: 3
E: 3

1) What is the first merge?

- ○ D and E: 6
- ○ B and D: 7
- ○ B and D and E: 10

2) What is the second merge?

- ○ B and D: 7
- ○ DE and B: 10
- ○ C and A: 90

3) What is the third merge?

- ○ DEB and C: 40
- ○ DEB and C: 50

4) What is the fourth merge?

- ○ None
- ○ DEBC and A: 100

5) What is the fifth merge?

- ○ None
- ○ DEBCA and F

6) What is the code for A?

- ○ 0
- ○ 1

7) What is the code for C?

- 1
- 01
- 10

8) What is the code for B?

- 001
- 110

9) What is the code for D?

- 1110
- 1111

10) What is the code for E?

- 1110
- 1111

11) 5 unique characters (A, B, C, D, E) can each be uniquely encoded in 3 bits (like 000, 001, 010, 011, and 100). With such a fixed-length code, how many bits are needed for the 100-character text?

- 100
- 300

12) For the Huffman code determined in the above questions, the number of bits per character is A: 1, C: 2, B: 3, D: 4, and E: 4. Recalling the frequencies in the instructions, how many bits are needed for the 100-character text?

- 14
- 166
- 300

Note: For Huffman encoded data, the dictionary must be included along with the compressed data, to enable decompression. That dictionary adds to the total bits used. However, typically only large data files get compressed, so the dictionary is usually a tiny fraction of the total size.

Building a Huffman tree

The data members in a Huffman tree node depend on the node type.

- Leaf nodes have two data members: a character from the input and an integer frequency for that character.
- Internal nodes have left and right child nodes, along with an integer frequency value that represents the sum of the left and right child frequencies.

A Huffman tree can be built from a character frequency table.

PARTICIPATION ACTIVITY | 8.11.7: Building a Huffman tree.

Animation content:

`undefined`

Animation captions:

1. The character frequency table is built for the input string, BANANAS.
2. A leaf node is built for each table entry and enqueued in a priority queue. Lower frequencies have higher priority.
3. Leaf nodes for S and B are removed from the queue. A parent is built with the sum of the frequencies and is enqueued into the priority queue.
4. The two nodes with frequency 2 are dequeued and the parent with frequency 2 + 2 = 4 is built.
5. The remaining 2 nodes are dequeued and given a parent.
6. When the priority queue has 1 node remaining, that node is the tree's root. The root is dequeued and returned.

PARTICIPATION ACTIVITY | 8.11.8: HuffmanBuildTree function.

Assume `HuffmanBuildTree("zyBooks")` is called.

1) The character frequency table has _____ entries.

- 5
- 6
- 7

2) The leaf node at the back of the priority queue, `nodes`, before the while loop begins is _____.

- z | 1
- B | 1
- o | 2

3) The parent node for nodes B and k has a frequency of _____.

- 1
- 2
- 4

Implementing with 1 node structure

Implementations commonly use the same node structure for leaf and internal nodes, instead of two distinct structures. Each node has a frequency, character, and 2 child pointers. The child pointers are set to null for leaves and the character is set to 0 for internal nodes.

Getting Huffman codes

Huffman codes for each character are built from a Huffman tree. Each character corresponds to a leaf node. The Huffman code for a character is built by tracing a path from the root to that character's leaf node, appending 0 when branching left or 1 when branching right.

PARTICIPATION ACTIVITY 8.11.9: Getting Huffman codes.

Animation content:

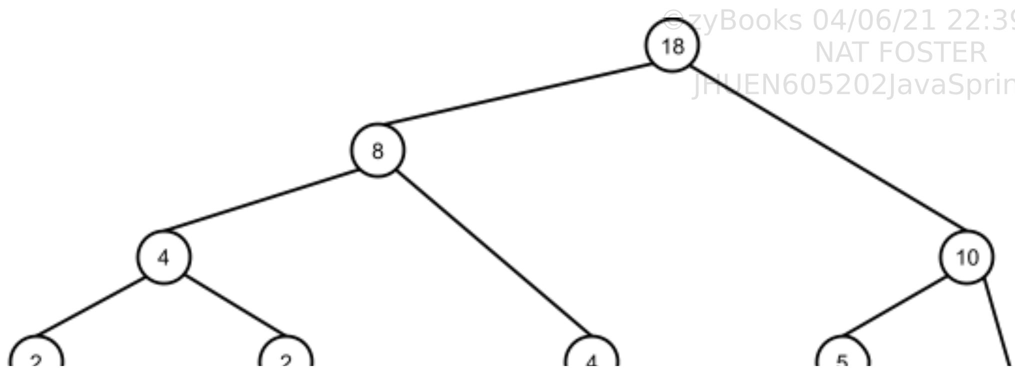
`undefined`

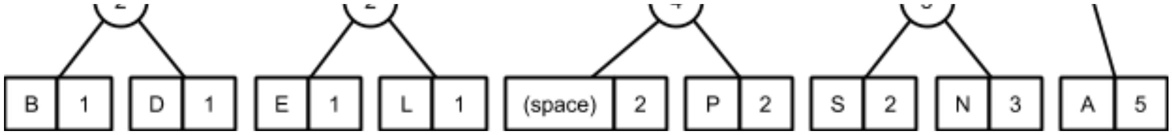
Animation captions:

1. Huffman codes are built from a Huffman tree. Left branches add a 0 to the code, right branches add a 1.
2. HuffmanGetCodes starts at the root node with an empty prefix.
3. A recursive call is made on the root's left child. The node is a leaf and A's code is set to the current prefix, 0.
4. The first recursive call completes. The next recursive call is made for node 4 and a prefix of "1".
5. Node 4 and node 2 are not leaves, so additional recursive calls are made.
6. B's code is set to 100.
7. The remaining recursive calls set codes for S and N.
8. Each distinct character has a code. Characters B and S have lower frequencies than A and N and thus have longer codes.

PARTICIPATION ACTIVITY 8.11.10: Huffman codes.

Below is the Huffman tree for "APPLES AND BANANAS"





1) What is the Huffman code for A?

- 10
- 11
- 101

2) What is the Huffman code for P?

- 01
- 0000
- 011

3) What is the length of the longest Huffman code?

- 3
- 4
- 5

Compressing data

To compress an input string, the Huffman codes are first obtained for each character. Then each character of the input string is processed, and corresponding bit codes are concatenated to produce the compressed result.

Figure 8.11.1: HuffmanCompress function.

```
HuffmanCompress(inputString) {
   // Build the Huffman tree
   root = HuffmanBuildTree(inputString)

   // Get the compression codes from the tree
   codes = HuffmanGetCodes(root, "", new Dictionary())

   // Build the compressed result
   result = ""
   for c in inputString {
      result += codes[c]
   }
   return result
}
```


PARTICIPATION ACTIVITY | 8.11.11: Compressing data.

Match each compression call to the result. Spaces are added between character codes for clarity, but would not exists in the actual compressed data.

HuffmanCompress("BANANAS") **HuffmanCompress("aabbbac")**

HuffmanCompress("zyBooks")

100 0 11 0 11 0 101

00 101 010 11 11 011 100

11 11 0 0 0 11 10

Reset

Decompressing Huffman coded data

To decompress Huffman code data, one can use a Huffman tree and trace the branches for each bit, starting at the root. When the final node of the branch is reached, the result has been found. The process continues until the entire item is decompressed.

PARTICIPATION ACTIVITY | 8.11.12: Decompressing Huffman code.

Animation captions:

1. The Huffman code is decompressed by first starting at the root. The branches are followed for each bit.
2. When the final node of the branch is reached, the result has been found.
3. Once the final node is reached decoding restarts at the root node.
4. The process continues until the entire item is decompressed.

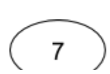
Figure 8.11.2: HuffmanDecompress function.

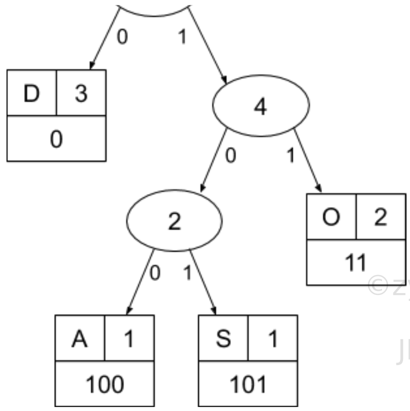
```
HuffmanDecompress(compressedString, treeRoot) {
   node = treeRoot
   result = ""
   for (bit in compressedString) {
      // Go to left or right child based on bit value
      if (bit == 0)
         node = node→left
      else
         node = node→right

      // If the node is a leaf, add the character to the
      // decompressed result and go back to the root node
      if (node is a leaf) {
         result += node→character
         node = treeRoot
      }
   }
   return result
}
```


PARTICIPATION ACTIVITY | 8.11.13: Decompressing Huffman code.

Use the tree below to decompress 0111101000101.





1) What is the first decoded character?

Check Show answer

2) What is the second decoded character?

Check Show answer

3) 11 yields the third character O.
0 yields the fourth character D.

What is the next decoded character?

Check Show answer

4) What is the decoded text?

Check Show answer