# 13.1 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 times. Each iteration moves the $i^{th}$ largest element into sorted position. The inner j-loop iterates through all adjacent pairs, comparing and swapping adjacent elements as needed, except for the last i pairs that are already in the correct position,.



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

Figure 13.1.1: 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 13.1.1: Bubble sort.

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

- True
- False

2) Bubble sort only swaps adjacent elements.

- True
- False



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

- True

○ False

How was this section? Provide feedback

# 13.2 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 13.2.1: 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 13.2.1: 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

How was this section? Provide feedback

# 13.3 Bucket sort

***Bucket sort*** is a numerical sorting algorithm that distributes numbers into buckets, sorts each bucket with an additional sorting algorithm, and then concatenates buckets together to build the sorted result. A ***bucket*** is a container for numerical values in a specific range. Ex: All numbers in the range 0 to 50 may be stored in a bucket representing this range. Bucket sort is designed for arrays with non-negative numbers.

Bucket sort first creates a list of buckets, each representing a range of numerical values. Collectively, the buckets represent the range from 0 to the maximum value in the array. For $N$ buckets and a maximum value of $M$, each bucket represents $\frac{M}{N}$ values. Ex: For 10 buckets and a maximum value of 50, each bucket represents a range of 5 value; the first bucket will hold values ranging from 0 to 4, the second bucket 5 to 9, and so on. Each array element is placed in the appropriate bucket. The bucket index is calculated as $\left\lfloor number * \frac{N-1}{M} \right\rfloor$. Then, each bucket is sorted with an additional sorting algorithm. Lastly, all buckets are concatenated together in order, and copied to the original array.

## Figure 13.3.1: Bucket sort algorithm.

```
BucketSort(numbers, numbersSize, bucketCount) {
   if (numbersSize < 1)
      return

   buckets = Create list of bucketCount buckets

   // Find the maximum value
   maxValue = numbers[0]
   for (i = 1; i < numbersSize; i++) {
      if (numbers[i] > maxValue)
         maxValue = numbers[i]
   }

   // Put each number in a bucket
   for each (number in numbers) {
      index = floor(number * (bucketCount - 1) / maxValue)
      Append number to buckets[index]
   }

   // Sort each bucket
   for each (bucket in buckets)
      Sort(bucket)

   // Combine all buckets back into numbers list
   result = Concatenate all buckets together
   Copy result to numbers
}
```

**PARTICIPATION ACTIVITY** 13.3.1: Bucket sort.

Suppose BucketSort is called to sort the list (71, 22, 99, 7, 14), using 5 buckets.

1) 71 and 99 will be placed into the same bucket

- True
- False

2) No bucket will have more than 2 numbers.

- True
- False

3) If 10 buckets were used instead of 5, no bucket would have more than 1 number.

- True
- False

Bucket sort terminology

The term "bucket sort" is sometimes used to refer to a category of sorting algorithms, instead of a specific sorting algorithm. When used as a categorical term, bucket sort refers to a sorting algorithm that places numbers into buckets based on some common attribute, and then combines bucket contents to produce a sorted array.

How was this section? Provide feedback

# 13.4 List data structure

A common approach for implementing a linked list is using two data structures:

1. List data structure: A ***list data structure*** is a data structure containing the list's head and tail, and may also include additional information, such as the list's size.
2. List node data structure: The list node data structure maintains the data for each list element, including the element's data and pointers to the other list element.

A list data structure is not required to implement a linked list, but offers a convenient way to store the list's head and tail. When using a list data structure, functions that operate on a list can use a single parameter for the list's data structure to manage the list.

A linked list can also be implemented without using a list data structure, which minimally requires using separate list node pointer variables to keep track of the list's head.

**PARTICIPATION ACTIVITY** 13.4.1: Linked lists can be stored with or without a list data structure.

### Animation content:

`undefined`

### Animation captions:

1. A linked list can be maintained without a list data structure, but a pointer to the head and tai of the list must be stored elsewhere, often as local variables.

2. A list data structure stores both the head and tail pointers in one object.

PARTICIPATION ACTIVITY 13.4.2: Linked list data structure.

1) A linked list must have a list data structure.

- True
- False

2) A list data structure can have additional information besides the head and tail pointers.

- True
- False

3) A linked list has O(n) space complexity, whether a list data structure is used or not.

- True
- False

How was this section? **Provide feedback**

# 13.5 Circular lists

A ***circular linked list*** is a linked list where the tail node's next pointer points to the head of the list, instead of null. A circular linked list can be used to represent repeating processes. Ex: Ocean water evaporates, forms clouds, rains down on land, and flows through rivers back into the ocean. The head of a circular linked list is often referred to as the *start* node.

A traversal through a circular linked list is similar to traversal through a standard linked list, but must terminate after reaching the head node a second time, as opposed to terminating when reaching null.

PARTICIPATION ACTIVITY 13.5.1: Circular list structure and traversal.

## Animation captions:

1. In a circular linked list, the tail node's next pointer points to the head node.
2. A circular doubly-linked list's head node's previous pointer points to the tail node.
3. Instead of stopping when the "current" pointer is null, traversal through a circular list stops when current comes back to the head node.

**PARTICIPATION ACTIVITY** 13.5.2: Circular list concepts.

1) Only a doubly-linked list can be circular.

- True
- False

2) In a circular doubly-linked list with at least 2 nodes, where does the head node's previous pointer point to?

- List head
- List tail
- null

3) In a circular linked list with at least 2 nodes, where does the tail node's next pointer point to?

- List head
- List tail
- null

4) In a circular linked list with 1 node, the tail node's next pointer points to the tail.

- True
- False

5) The following code can be used to traverse a circular, doubly-linked list in

reverse order.

```
CircularListTraverseReverse(tail) {
   current = tail
   do {
      visit current
      current = current->previous
   } while (current != tail)
}
```

- True
- False

How was this section? Provide feedback