3.1 Introduction to algorithms

Algorithms

An ***algorithm*** describes a sequence of steps to solve a computational problem or perform a calculation. An algorithm can be described in English, pseudocode, a programming language, hardware, etc. A ***computational problem*** specifies an input, a question about the input that can be answered using a computer, and the desired output.

PARTICIPATION ACTIVITY 3.1.1: Computational problems and algorithms.

Animation captions:

1. A computational problem is a problem that can be solved using a computer. A computational problem specifies the problem input, a question to be answered, and the desired output.
2. For the problem of finding the maximum value in an array, the input is an array of numbers.
3. The problem's question is: What is the maximum value in the input array? The problem's output is a single value that is the maximum value in the array.
4. The FindMax algorithm defines a sequence of steps that determines the maximum value in the array.

PARTICIPATION ACTIVITY 3.1.2: Algorithms and computational problems.

Consider the problem of determining the number of times (or frequency) a specific word appears in a list of words.

1) Which can be used as the problem input?

- ○ String for user-specified word
- ○ Array of unique words and string for user-specified word
- ○ Array of all words in the list and string for user-specified word

2) What is the problem output?

- ○ Integer value for the frequency of most frequent word

○ String value for the most frequent word in input array

○ Integer value for the frequency of specified word

3) An algorithm to solve this computation problem must be written using a programming language.

○ True

○ False

Practical applications of algorithms

Computational problems can be found in numerous domains, including e-commerce, internet technologies, biology, manufacturing, transportation, etc. Algorithms have been developed for numerous computational problems within these domains.

A computational problem can be solved in many ways, but finding the best algorithm to solve a problem can be challenging. However, many computational problems have common subproblems, for which efficient algorithms have been developed. The examples below describe a computational problem within a specific domain and list a common algorithm (each discussed elsewhere) that can be used to solve the problem.

Table 3.1.1: Example computational problems and common algorithms.

Application domain	Computational problem	Common algorithm
DNA analysis	Given two DNA sequences from different individuals, what is the longest shared sequence of nucleotides?	*Longest common substring problem:* A longest common substring algorithm determines the longest common substring that exists in two input strings. DNA sequences can be represented using strings consisting of the letters A, C, G, and T to represent the four different nucleotides.
Search engines	Given a product ID and a sorted array of all in-stock products, is the product in	*Binary search:* The binary search algorithm is an efficient algorithm for searching a list. The list's elements must be sorted and directly accessible (such as an array).

	stock and what is the product's price?	
Navigation	Given a user's current location and desired location, what is the fastest route to walk to the destination?	*Dijkstra's shortest path:* Dijkstra's shortest path algorithm determines the shortest path from a start vertex to each vertex in a graph. The possible routes between two locations can be represented using a graph, where vertices represent specific locations and connecting edges specify the time required to walk between those two locations.

PARTICIPATION ACTIVITY | 3.1.3: Computational problems and common algorithms.

Match the common algorithm to another computational problem that can be solved using that algorithm.

If unable to drag and drop, refresh the page.

Binary search **Shortest path algorithm** **Longest common substring**

	Do two student essays share a common phrase consisting of a sequence of more than 100 letters?
	Given the airports at which an airline operates and distances between those airports, what is the shortest total flight distance between two airports?
	Given a sorted list of a company's employee records and an employee's first and last name, what is a specific employee's phone number?

Reset

Efficient algorithms and hard problems

Computer scientists and programmers typically focus on using and designing efficient algorithms to solve problems. Algorithm efficiency is most commonly measured by the algorithm runtime, and an efficient algorithm is one whose runtime increases no more than polynomially with respect to the input size. However, some problems exist for which an efficient algorithm is unknown.

NP-complete problems are a set of problems for which no known efficient algorithm exists. NP-complete problems have the following characteristics:

- No efficient algorithm has been found to solve an NP-complete problem.
- No one has proven that an efficient algorithm to solve an NP-complete problem is impossible.
- If an efficient algorithm exists for one NP-complete problem, then all NP-complete problems can be solved efficiently.

By knowing a problem is NP-complete, instead of trying to find an efficient algorithm to solve the problem, a programmer can focus on finding an algorithm to efficiently find a good, but non-optimal, solution.

PARTICIPATION ACTIVITY 3.1.4: Example NP-complete problem: Cliques.

Animation captions:

1. A programmer may be asked to write an algorithm to solve the problem of determining if a set of K people who all know each other exists within a graph of a social network?
2. For the example social network graph and K = 3, the algorithm should return yes. Xiao, Sean, and Tanya all know each other. Sean, Tanya, and Eve also all know each other.
3. For K = 4, no set of 4 individual who all know each other exists, and the algorithm, should return no.
4. This problem is equivalent to the clique decision problem, which is NP-complete, and no known polynomial time algorithm exists.

PARTICIPATION ACTIVITY 3.1.5: Efficient algorithm and hard problems.

1) An algorithm with a polynomial runtime is considered efficient.

- ○ True
- ○ False

2) An efficient algorithm exists for all computational problems.

- ○ True

○ False

3) An efficient algorithm to solve an NP-complete problem may exist.

○ True

○ False

3.2 Relation between data structures and algorithms

Algorithms for data structures

Data structures not only define how data is organized and stored, but also the operations performed on the data structure. While common operations include inserting, removing, and searching for data, the algorithms to implement those operations are typically specific to each data structure. Ex: Appending an item to a linked list requires a different algorithm than appending an item to an array.

PARTICIPATION ACTIVITY 3.2.1: A list avoids the shifting problem.

Animation content:

undefined

Animation captions:

1. The algorithm to append an item to an array determines the current size, increases the array size by 1, and assigns the new item as the last array element.
2. The algorithm to append an item to a linked list points the tail node's next pointer and the list's tail pointer to the new node.

PARTICIPATION ACTIVITY 3.2.2: Algorithms for data structures.

Consider the array and linked list in the animation above. Can the following algorithms be implemented with the same code for both an array and linked list?

1) Append an item

Yes

No

2) Return the first item

Yes

No

3) Return the current size

Yes

No

Algorithms using data structures

Some algorithms utilize data structures to store and organize data during the algorithm execution. Ex: An algorithm that determines a list of the top five salespersons, may use an array to store salespersons sorted by their total sales.

Figure 3.2.1: Algorithm to determine the top five salespersons using an array.

```
DisplayTopFiveSalespersons(allSalespersons) {
   // topSales array has 5 elements
   // Array elements have subitems for name and total sales
   // Array will be sorted from highest total sales to lowest total sales
   topSales = Create array with 5 elements

   // Initialize all array elements with a negative sales total
   for (i = 0; i < topSales⇢length; ++i) {
      topSales[i]⇢name = ""
      topSales[i]⇢salesTotal = -1
   }

   for each salesPerson in allSalespersons {
      // If salesPerson's total sales is greater than the last
      // topSales element, salesPerson is one of the top five so far
      if (salesPerson⇢salesTotal > topSales[topSales⇢length -
1]⇢salesTotal) {

         // Assign the last element in topSales with the current
salesperson
         topSales[topSales⇢length - 1]⇢name =  salesPerson⇢name
         topSales[topSales⇢length - 1]⇢salesTotal =
salesPerson⇢salesTotal

         // Sort topSales in descending order
         SortDescending(topSales)
      }
   }

   // Display the top five salespersons
   for (i = 0; i < topSales⇢length; ++i) {
      Display topSales[i]
   }
}
```

PARTICIPATION ACTIVITY 3.2.3: Top five salespersons.

1) Which of the following is *not* equal to the number of items in the topSales array?

- ○ topSales⇢length
- ○ 5
- ○ allSalesperson⇢length

2) To adapt the algorithm to display the top 10 salespersons, what modifications are required?

- ○ Only the array creation

- ○ All loops in the algorithm
- ○ Both the creation and all loops

3) If allSalespersons only contains three elements, the DisplayTopFiveSalespersons algorithm will display two elements with no name and -1 sales.

- ○ True
- ○ False

3.3 Algorithm efficiency

Algorithm efficiency

An algorithm describes the method to solve a computational problem. Programmers and computer scientists should use or write efficient algorithms. ***Algorithm efficiency*** is typically measured by the algorithm's computational complexity. ***Computational complexity*** is the amount of resources used by the algorithm. The most common resources considered are the runtime and memory usage.

PARTICIPATION ACTIVITY 3.3.1: Computational complexity.

Animation captions:

1. An algorithm's computational complexity includes runtime and memory usage.
2. Measuring runtime and memory usage allows different algorithms to be compared.
3. Complexity analysis is used to identify and avoid using algorithms with long runtimes or high memory usage.

PARTICIPATION ACTIVITY 3.3.2: Algorithm efficiency and computational complexity.

1) Computational complexity analysis allows the efficiency of algorithms to be compared.

- ○ True
- ○ False

2) Two different algorithms that produce the same result have the same computational complexity.

   ○ True

   ○ False

3) Runtime and memory usage are the only two resources making up computational complexity.

   ○ True

   ○ False

Runtime complexity, best case, and worst case

An algorithm's ***runtime complexity*** is a function, T(N), that represents the number of constant time operations performed by the algorithm on an input of size N. Runtime complexity is discussed in more detail elsewhere.

Because an algorithm's runtime may vary significantly based on the input data, a common approach is to identify best and worst case scenarios. An algorithm's ***best case*** is the scenario where the algorithm does the minimum possible number of operations. An algorithm's ***worst case*** is the scenario where the algorithm does the maximum possible number of operations.

Input data size must remain a variable

A best case or worst case scenario describes contents of the algorithm's input data only. The input data size must remain a variable, N. Otherwise, the overwhelming majority of algorithms would have a best case of N=0, since no input data would be processed. In both theory and practice, saying "the best case is when the algorithm doesn't process any data" is not useful. Complexity analysis always treats the input data size as a variable.

PARTICIPATION ACTIVITY | 3.3.3: Linear search best and worst cases.

Animation captions:

1. LinearSearch searches through array elements until finding the key. Searching for 26 requires iterating through the first 3 elements.

2. The search for 26 is neither the best nor the worst case.
3. Searching for 54 only requires one comparison and is the best case: The key is found at the start of the array. No other search could perform fewer operations.
4. Searching for 82 compares against all array items and is the worst case: The number is not found in the array. No other search could perform more operations.

PARTICIPATION ACTIVITY 3.3.4: FindFirstLessThan algorithm best and worst case.

Consider the following function that returns the first value in a list that is less than the specified value. If no list items are less than the specified value, the specified value is returned.

```
FindFirstLessThan(list, listSize, value) {
   for (i = 0; i < listSize; i++) {
      if (list[i] < value)
         return list[i]
   }
   return value // no lesser value found
}
```

If unable to drag and drop, refresh the page.

Best case | Neither best nor worst case | Worst case

	No items in the list are less than `value`.
	The first half of the list has elements greater than `value` and the second half has elements less than `value`.
	The first item in the list is less than `value`.

Reset

PARTICIPATION ACTIVITY 3.3.5: Best and worst case concepts.

1) The linear search algorithm's best case scenario is when N = 0.

○ True

○ False

2) An algorithm's best and worst case scenarios are always different.

○ True

○ False

Space complexity

An algorithm's ***space complexity*** is a function, S(N), that represents the number of fixed-size memory units used by the algorithm for an input of size N. Ex: The space complexity of an algorithm that duplicates a list of numbers is S(N) = 2N + k, where k is a constant representing memory used for things like the loop counter and list pointers.

Space complexity includes the input data and additional memory allocated by the algorithm. An algorithm's ***auxiliary space complexity*** is the space complexity not including the input data. Ex: An algorithm to find the maximum number in a list will have a space complexity of S(N) = N + k, but an auxiliary space complexity of S(N) = k, where k is a constant.

PARTICIPATION ACTIVITY | 3.3.6: FindMax space complexity and auxiliary space complexity.

Animation content:

undefined

Animation captions:

1. FindMax's arguments represent input data. Non-input data includes variables allocated in the function body: maximum and i.
2. The list's size is a variable, N. Three integers are also used, listSize, maximum, and i, making the space complexity S(N) = N + 3.
3. The auxiliary space complexity includes only the non-input data, which does not increase for larger input lists.
4. The function's auxiliary space complexity is S(N) = 2.

PARTICIPATION ACTIVITY | 3.3.7: Space complexity of GetEvens function.

Consider the following function, which builds and returns a list of even numbers from the input list.

```
GetEvens(list, listSize) {
   i = 0
   evensList = Create new, empty list
   while (i < listSize) {
      if (list[i] % 2 == 0)
         Add list[i] to evensList
      i = i + 1
   }
   return evensList
}
```

1) What is the maximum possible size of the returned list?

- ○ listSize
- ○ listSize / 2

2) What is the minimum possible size of the returned list?

- ○ listSize / 2
- ○ 1
- ○ 0

3) What is the worst case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

- ○ S(N) = N + k
- ○ S(N) = k

4) What is the best case auxiliary space complexity of GetEvens if N is the list's size and k is a constant?

- ○ S(N) = N + k
- ○ S(N) = k