1.1 Data structures

Data structures

A ***data structure*** is a way of organizing, storing, and performing operations on data. Operations performed on a data structure include accessing or updating stored data, searching for specific data, inserting new data, and removing data. The following provides a list of basic data structures.

Table 1.1.1: Basic data structures.

Data structure	Description
Record	A ***record*** is the data structure that stores subitems, with a name associated with each subitem.
Array	An ***array*** is a data structure that stores an ordered list of items, with each item is directly accessible by a positional index.
Linked list	A ***linked list*** is a data structure that stores an ordered list of items in nodes, where each node stores data and has a pointer to the next node.
Binary tree	A ***binary tree*** is a data structure in which each node stores data and has up to two children, known as a left child and a right child.
Hash table	A ***hash table*** is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array.
Heap	A ***max-heap*** is a tree that maintains the simple property that a node's key is greater than or equal to the node's childrens' keys. A ***min-heap*** is a tree that maintains the simple property that a node's key is less than or equal to the node's childrens' keys.
Graph	A ***graph*** is a data structure for representing connections among items, and consists of vertices connected by edges. A ***vertex*** represents an item in a graph. An ***edge*** represents a connection between two vertices in a graph.

PARTICIPATION

ACTIVITY 1.1.1: Basic data structures.

1) A linked list stores items in an unspecified order.
   - True
   - False

2) A node in binary tree can have zero, one, or two children.
   - True
   - False

3) A list node's data can store a record with multiple subitems.
   - True
   - False

4) Items stored in an array can be accessed using a positional index.
   - True
   - False

Choosing data structures

The selection of data structures used in a program depends on both the type of data being stored and the operations the program may need to perform on that data. Choosing the best data structure often requires determining which data structure provides a good balance given expected uses. Ex: If a program requires fast insertion of new data, a linked list is a better choice than an array.

PARTICIPATION ACTIVITY 1.1.2: A list avoids the shifting problem.

Animation content:

```
undefined
```

Animation captions:

1. Inserting an item at a specific location in an array requires making room for the item by shifting higher-indexed items.

2. Once the higher index items have been shifted, the new item can be inserted at the desired index.
3. To insert new item in a linked list, a list node for the new item is first created.
4. Item B's next pointer is assigned to point to item C. Item A's next pointer is updated to point to item B. No shifting of other items was required.

PARTICIPATION ACTIVITY 1.1.3: Basic data structures.

1) Inserting an item at the end of a 999-item array requires how many items to be shifted?

Check Show answer

2) Inserting an item at the end of a 999-item linked list requires how many items to be shifted?

Check Show answer

3) Inserting an item at the beginning of a 999-item array requires how many items to be shifted?

Check Show answer

4) Inserting an item at the beginning of a 999-item linked list requires how many items to be shifted?

Check Show answer

1.2 Abstract data types

Abstract data types (ADTs)

An ***abstract data type*** (***ADT***) is a data type described by predefined user operations, such as "insert data at rear," without indicating how each operation is implemented. An ADT can be implemented using different underlying data structures. However, a programmer need not have knowledge of the underlying implementation to use an ADT.

Ex: A list is a common ADT for holding ordered data, having operations like append a data item, remove a data item, search whether a data item exists, and print the list. A list ADT is commonly implemented using arrays or linked list data structures.

PARTICIPATION ACTIVITY 1.2.1: List ADT using array and linked lists data structures.

Animation captions:

1. A new list named agesList is created. Items can be appended to the list. The items are ordered.
2. Printing the list prints the items in order.
3. A list ADT is commonly implemented using array and linked list data structures. But, a programmer need not have knowledge of which data structure is used to use the list ADT.

PARTICIPATION ACTIVITY 1.2.2: Abstract data types.

1) Starting with an empty list, what is the list contents after the following operations?

   Append(list, 11)
   Append(list, 4)
   Append(list, 7)

   - 4, 7, 11
   - 7, 4, 11
   - 11, 4, 7

2) A remove operation for a list ADT will remove the specified item. Given a

list with contents: 2, 20, 30, what is the list contents after the following operation?

Remove(list, item 2)

- 2, 30
- 2, 20, 30
- 20, 30

3) A programmer must know the underlying implementation of the list ADT in order to use a list.

- True
- False

4) A list ADT's underlying data structure has no impact on the program's execution.

- True
- False

Common ADTs

Table 1.2.1: Common ADTs.

Abstract data type	Description	Common underlying data structures
List	A ***list*** is an ADT for holding ordered data.	Array, linked list
Dynamic array	A ***dynamic array*** is an ADT for holding ordered data and allowing indexed access.	Array
Stack	A ***stack*** is an ADT in which items are only inserted on or removed from the top of a stack.	Linked list
Queue	A ***queue*** is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.	Linked list

Deque	A ***deque*** (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back.	Linked list
Bag	A ***bag*** is an ADT for storing items in which the order does not matter and duplicate items are allowed.	Array, linked list
Set	A ***set*** is an ADT for a collection of distinct items.	Binary search tree, hash table
Priority queue	A ***priority queue*** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority.	Heap
Dictionary (Map)	A ***dictionary*** is an ADT that associates (or maps) keys with values.	Hash table, binary search tree

PARTICIPATION ACTIVITY 1.2.3: Common ADTs.

Consider the ADTs listed in the table above. Match the ADT with the description of the order and uniqueness of items in the ADT.

Set **Bag** **Priority queue** **List**

	Items are ordered based on how items are added. Duplicate items are allowed.
	Items are not ordered. Duplicate items are not allowed.
	Items are ordered based on items' priority. Duplicate items are allowed.
	Items are not ordered. Duplicate items are allowed.

Reset

1.3 Applications of ADTs

Abstraction and optimization

Abstraction means to have a user interact with an item at a high-level, with lower-level internal details hidden from the user. ADTs support abstraction by hiding the underlying implementation details and providing a well-defined set of operations for using the ADT.

Using abstract data types enables programmers or algorithm designers to focus on higher-level operations and algorithms, thus improving programmer efficiency. However, knowledge of the underlying implementation is needed to analyze or improve the runtime efficiency.

PARTICIPATION ACTIVITY 1.3.1: Programming using ADTs.

Animation captions:

1. Abstraction simplifies programming. ADTs allow programmers to focus on choosing which ADTs best match a program's needs.
2. Both the List and Queue ADTs support efficient interfaces for removing items from one end (removing oldest entry) and adding items to the other end (adding new entries).
3. The list ADT supports printing the list contents, but the queue ADT does not.
4. To use the List (or Queue) ADT, the programmer does not need to know the List's underlying implementation.

PARTICIPATION ACTIVITY 1.3.2: Programming with ADTs.

Consider the example in the animation above.

1) The Queue ADT _____.

- cannot be used to implement the program requirements
- does not provide the best abstraction for the program requirements

2) The list ADT _____.

- can only be implemented using an array
- can only be implemented using a linked list
- can be implemented in numerous ways

3) Knowledge of an ADT's underlying implementation is needed to analyze the runtime efficiency.

- True
- False

ADTs in standard libraries

Most programming languages provide standard libraries that implement common abstract data types. Some languages allow programmers to choose the underlying data structure used for the ADTs. Other programming languages may use a specific data structure to implement each ADT, or may automatically choose the underlying data-structure.

Table 1.3.1: Standard libraries in various programming languages.

Programming language	Library	Common supported ADTs
Python	Python standard library	list, set, dict, deque
C++	Standard template library (STL)	vector, list, deque, queue, stack, set, map
Java	Java collections framework (JCF)	Collection, Set, List, Map, Queue, Deque

PARTICIPATION ACTIVITY 1.3.3: ADTs in standard libraries.

1) Python, C++, and Java all provide built-in support for a deque ADT.

True

False

2) The underlying data structure for a list data structure is the same for all programming languages.

- True
- False

3) ADTs are only supported in standard libraries.

- True
- False

1.4 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 1.4.1: 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 1.4.2: 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 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 1.4.1: 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 inputs 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 stock and what is the product's price?	*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).
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 1.4.3: Computational problems and common algorithms.

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

Shortest path algorithm | Binary search | 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 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 problem 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 1.4.4: 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 exits, 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 1.4.5: 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

1.5 Heuristics

Heuristics

In practice, solving a problem in the optimal or most accurate way may require more computational resources than are available or feasible. Algorithms implemented for such problems often use a ***heuristic***: A technique that willingly accepts a non-optimal or less accurate solution in order to improve execution speed.

PARTICIPATION ACTIVITY 1.5.1: Introduction to the knapsack problem.

Animation content:

undefined

Animation captions:

1. A knapsack is a container for items, much like a backpack or bag. Suppose a particular knapsack can carry at most 30 pounds worth of items.
2. Each item has a weight and value. The goal is to put items in the knapsack such that the weight <= 30 pounds and the value is maximized.
3. Taking a 20 pound item with an 8 pound item is an option, worth $142.
4. If more than 1 of each item can be taken, 2 of item 1 and 1 of item 4 provide a better option, worth $145.
5. Trying all combinations will give an optimal answer, but is time consuming. A heuristic algorithm may choose a simpler, but non-optimal approach.

PARTICIPATION ACTIVITY 1.5.2: Heuristics.

1) A heuristic is a way of producing an optimal solution to a problem.
   - ○ True
   - ○ False

2) A heuristic technique used for numerical computation may sacrifice accuracy to gain speed.
   - ○ True
   - ○ False

PARTICIPATION ACTIVITY 1.5.3: The knapsack problem.

Refer to the example in the animation above.

1) Which of the following options provides the best value?
   - 5 6-pound items
   - 2 6-pound items and 1 18-pound item
   - 3 8-pound items and 1 6-pound item.

2) The optimal solution has a value of $162 and has one of each item: 6-pound, 8-pound, and 18-pound.
   - True
   - False

3) Which approach would guarantee finding an optimal solution?
   - Taking the largest item that fits in the knapsack repeatedly until no more items will fit.
   - Taking the smallest item repeatedly until no more items will fit in the knapsack.
   - Trying all combinations of items and picking the one with maximum value.

Heuristic optimization

A ***heuristic algorithm*** is an algorithm that quickly determines a near optimal or approximate solution. Such an algorithm can be designed to solve the ***0-1 knapsack problem***: The knapsack problem with the quantity of each item limited to 1.

A heuristic algorithm to solve the 0-1 knapsack problem can choose to always take the most valuable item that fits in the knapsack's remaining space. Such an algorithm uses the heuristic of choosing the highest value item, without considering the impact on the remaining choices. While the algorithm's simple choices are aimed at optimizing the total value, the final result may not be optimal.

PARTICIPATION ACTIVITY 1.5.4: Non-optimal, heuristic algorithm to solve the 0-1 knapsack.

Animation content:

```
undefined
```

Animation captions:

1. The item list is sorted and the most valuable item is put into the knapsack first.
2. No remaining items will fit in the knapsack.
3. The resulting value of $95 is inferior to taking the 12 and 8 pound items, collectively worth $102.
4. The heuristic algorithm sacrifices optimality for efficiency and simplicity.

PARTICIPATION ACTIVITY 1.5.5: Heuristic algorithm and the 0-1 knapsack problem.

1) Which is not commonly sacrificed by a heuristic algorithm?
   - speed
   - optimality
   - accuracy

2) What restriction does the 0-1 knapsack problem have, in comparison with the regular knapsack problem?
   - The knapsack's weight limit cannot be exceeded.
   - At most 1 of each item can be taken.
   - The value of each item must be less than the item's weight.

3) Under what circumstance would the Knapsack01 function not put the most valuable item into the knapsack?
   - The item list contains only 1 item.
   - The weight of the most valuable item is greater than the knapsack's limit.

Self-adjusting heuristic

A ***self-adjusting heuristic*** is an algorithm that modifies a data structure based on how that data structure is used. Ex: Many self-adjusting data structures, such as red-black trees and AVL trees, use a self-adjusting heuristic to keep the tree balanced. Tree balancing organizes data to allow for faster access.

Ex: A self-adjusting heuristic can be used to speed up searches for frequently-searched-for list items by moving a list item to the front of the list when that item is searched for. This heuristic is self-adjusting because the list items are adjusted when an search is performed.

PARTICIPATION ACTIVITY 1.5.6: Move-to-front self-adjusting heuristic.

Animation content:

undefined

Animation captions:

1. 42 is at the end of a list with 8 items. A linear search for 42 compares against 8 items.
2. The move-to-front heuristic moves 42 to the front after the search.
3. Another search for 42 now only requires 1 comparison. 42 is left at the front of the list.
4. A search for 64 compares against 8 items and moves 64 to the front of the list.
5. 42 is no longer at the list's front, but a search for 42 need only compare against 2 items.

PARTICIPATION ACTIVITY 1.5.7: Move-to-front self-adjusting heuristic.

Suppose a move-to-front heuristic is used on a list that starts as (56, 11, 92, 81, 68, 44).

1) A first search for 81 compares against how many list items?
   - ○ 0
   - ○ 3
   - ○ 4

2) A subsequent search for 81 compares against how many list items?
   - ○ 1
   - ○ 2
   - ○ 4

3) Which scenario results in faster searches?

- ○ Back-to-back searches for the same key.
- ○ Every search is for a key different than the previous search.

1.6 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 1.6.1: 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 1.6.2: 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 1.6.1: 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
   Create topSales 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]→totalSales =  salesPerson→totalSales

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

   // Display the top five salespersons
```

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

PARTICIPATION ACTIVITY 1.6.3: 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 salesperson, 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 elements with no name and negative sales.
   - True
   - False

1.7 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 1.7.1: 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 1.7.2: 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 1.7.3: 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 1.7.4: 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
}
```

Best case | Worst case | Neither best nor 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 1.7.5: Best and worst case concepts.

1) Nearly every algorithm has a best case time complexity 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 1.7.6: FindMax space complexity and auxiliary space complexity.

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, 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 1.7.7: 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 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

1.8 Constant time operations

Constant time operations

In practice, designing an efficient algorithm aims to lower the amount of time that an algorithm runs. However, a single algorithm can always execute more quickly on a faster processor. Therefore, the theoretical analysis of an algorithm describes runtime in terms of number of constant time operations, not nanoseconds. A ***constant time operation*** is an operation that, for a given processor, always operates in the same amount of time, regardless of input values.

PARTICIPATION ACTIVITY 1.8.1: Constant time vs. non-constant time operations.

Animation captions:

1. x = 10, y = 20, a = 1000, and b = 2000 are assigning variables with values, and are constant time operations.
2. A CPU multiplies values 10 and 20 at the same speed as 1000 and 2000. Multiplication is a constant time operation.
3. A loop that iterates x times, adding y to a sum each iteration, will take longer if x is larger. The loop is not constant time.
4. String concatenation is another common operation that is not constant time, because more characters need to be copied for larger strings.

PARTICIPATION ACTIVITY 1.8.2: Constant time operations.

1) The statement below that assigns x with y is a constant time operation.

```
y = 10
x = y
```

- True
- False

2) A loop is never a constant time operation.

- True
- False

3) The 3 constant time operations in the code below can collectively be considered 1 constant time operation.

```
x = 26.5
y = 15.5
z = x + y
```

- True
- False

Identifying constant time operations

The programming language being used, as well as the hardware running the code, both affect what is and what is not a constant time operation. Ex: Most modern processors perform arithmetic operations on integers and floating point values at a fixed rate that is unaffected by operand values. Part of the reason for this is that the floating point and integer values have a fixed size. The table below summarizes operations that are generally considered constant time operations.

Table 1.8.1: Common constant time operations.

Operation	Example

Addition, subtraction, multiplication, and division of fixed size integer or floating point values.	`w = 10.4`
	`x = 3.4`
	`y = 2.0`
	`z = (w - x) / y`

Assignment of a reference, pointer, or other fixed size data value.	`x = 1000`
	`y = x`
	`a = true`
	`b = a`

Comparison of two fixed size data values.	`a = 100` `b = 200`

	`if (b > a) { ... }`
Read or write an array element at a particular index.	`x = arr[index]` `arr[index + 1] = x + 1`

PARTICIPATION ACTIVITY 1.8.3: Identifying constant time operations.

1) In the code below, suppose str1 is a pointer or reference to a string. The code only executes in constant time if the assignment copies the pointer/reference, and not all the characters in the string.

```
str2 = str1
```

- True
- False

2) Certain hardware may execute division more slowly than multiplication, but both may still be constant time operations.

- True
- False

3) The hardware running the code is the only thing that affects what is and what is not a constant time operation.

- True
- False

1.9 Growth of functions and complexity

Upper and lower bounds

An algorithm with runtime complexity T(N) has a lower bound and an upper bound.

- ***Lower bound***: A function f(N) that is ≤ the best case T(N), for all values of N ≥ 1.
- ***Upper bound***: A function f(N) that is ≥ the worst case T(N), for all values of N ≥ 1.

Ex: An algorithm with best case runtime $T(N) = 7N + 36$ and worst case runtime $T(N) = 3N^2 + 10N + 17$, has a lower bound $f(N) = 7N$ and an upper bound $f(N) = 30N^2$. These lower and upper bounds provide a general picture of the runtime, while using simpler functions than the exact runtime.

Upper and lower bounds in the context of runtime complexity

In strict mathematical terms, upper and lower bounds do not relate to best or worst case runtime complexities. In the context of algorithm complexity analysis, the terms are often used to collectively bound all runtime complexities for an algorithm. Therefore, the terms are used in this section such that the upper bound is ≥ the worst case T(N) and the lower bound is ≤ the best case T(N).

PARTICIPATION ACTIVITY 1.9.1: Upper and lower bounds.

Animation content:

undefined

Animation captions:

1. An algorithm's worst and best case runtimes are represented by the blue and purple curves, respectively.
2. $2N^2$, shown in yellow, is a lower bound. The lower bound is less than or equal to both runtime functions for all N ≥ 1.
3. $30N^2$, shown in orange, is an upper bound. The upper bound is greater than or equal to both runtime functions for all N ≥ 1.
4. Together, the upper and lower bounds enclose all possible runtimes for this algorithm.

PARTICIPATION ACTIVITY 1.9.2: Upper and lower bounds.

Suppose an algorithm's best case runtime complexity is $T(N) = 3N + 6$, and the algorithm's worst case runtime is $T(N) = 5N^2 + 7N$.

1) Which function is a lower bound for the algorithm?
   - $5N^2$
   - $5N$
   - $3N$

2) Which function is an upper bound for the algorithm?
   - $12N^2$
   - $5N^2$
   - $7N$

3) $5N^2 + 7N$ is an upper bound for the algorithm.
   - True
   - False

4) $3N + 6$ is a lower bound for the algorithm.
   - True
   - False

Growth rates and asymptotic notations

An additional simplification can factor out the constant from a bounding function, leaving a function that categorizes the algorithm's growth rate. Ex: Instead of saying that an algorithm's runtime function has an upper bound of $30N^2$, the algorithm could be described as having a worst case growth rate of N^2. ***Asymptotic notation*** is the classification of runtime complexity that uses functions that indicate only the growth rate of a bounding function. Three asymptotic notations are commonly used in complexity analysis:

- ***O notation*** provides a growth rate for an algorithm's upper bound.
- ***Ω notation*** provides a growth rate for an algorithm's lower bound.
- ***Θ notation*** provides a growth rate that is both an upper and lower bound.

Table 1.9.1: Notations for algorithm complexity analysis.

Notation	General form	Meaning
O	$T(N) = O(f(N))$	A positive constant c exists such that, for all N ≥ 1, $T(N) \leq c * f(N)$.
Ω	$T(N) = \Omega(f(N))$	A positive constant c exists such that, for all N ≥ 1, $T(N) \geq c * f(N)$.
Θ	$T(N) = \Theta(f(N))$	$T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

PARTICIPATION ACTIVITY 1.9.3: Asymptotic notations.

Suppose $T(N) = 2N^2 + N + 9$.

1) $T(N) = O(N^2)$
   - True
   - False

2) $T(N) = \Omega(N^2)$
   - True
   - False

3) $T(N) = \Theta(N^2)$
   - True
   - False

4) $T(N) = O(N^3)$
   - True
   - False

5) $T(N) = \Omega(N^3)$
   - True
   - False

1.10 O notation

1.10 O notation

Big O notation

Big O notation is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size. In Big O notation, all functions that have the same growth rate (as determined by the highest order term of the function) are characterized using the same Big O notation. In essence, all functions that have the same growth rate are considered equivalent in Big O notation.

Given a function that describes the running time of an algorithm, the Big O notation for that function can be determined using the following rules:

1. If f(N) is a sum of several terms, the highest order term (the one with the fastest growth rate) is kept and others are discarded.
2. If f(N) has a term that is a product of several factors, all constants (those that are not in terms of N) are omitted.

PARTICIPATION ACTIVITY 1.10.1: Determining Big O notation of a function.

Animation captions:

1. Determine a function that describes the running time of the algorithm, and then compute the Big O notation of that function.
2. Apply rules to obtain the Big O notation of the function.
3. All functions with the same growth rate are considered equivalent in Big O notation.

PARTICIPATION ACTIVITY 1.10.2: Big O notation.

1) Which of the following Big O notations is equivalent to O(N+9999)?

- O(1)
- O(N)
- O(9999)

2) Which of the following Big O notations is equivalent to O(734·N)?

- O(N)
- O(734)
- $O(N^2)$

- O(734·N^2)

3) Which of the following Big O notations is equivalent to O(12·N +6·N^3 + 1000)?

- O(1000)
- O(N)
- O(N^3)

Big O notation of composite functions

The following rules are used to determine the Big O notation of composite functions: c denotes a constant

Figure 1.10.1: Rules for determining Big O notation of composite functions.

Composite function	Big O notation
c · O(f(N))	O(f(N))
c + O(f(N))	O(f(N))
g(N) · O(f(N))	O(g(N) · f(N))
g(N) + O(f(N))	O(g(N) + f(N))

PARTICIPATION ACTIVITY 1.10.3: Big O notation for composite functions.

Determine the simplified Big O notation.

1) 10 · O(N^2)

- O(10)
- O(N^2)
- O(10 · N^2)

2) 10 + O(N^2)

- O(10)

- O(N^2)
- O($10 + N^2$)

3) $3 \cdot N \cdot O(N^2)$

- O(N^2)
- O($3 \cdot N^2$)
- O(N^3)

4) $2 \cdot N^3 + O(N^2)$

- O(N^2)
- O(N^3)
- O($N^2 + N^3$)

5) $log_2 N$

- O($log_2 N$)
- O($log N$)
- O(N)

Runtime growth rate

One consideration in evaluating algorithms is that the efficiency of the algorithm is most critical for large input sizes. Small inputs are likely to result in fast running times because N is small, so efficiency is less of a concern. The table below shows the runtime to perform f(N) instructions for different functions f and different values of N. For large N, the difference in computation time varies greatly with the rate of growth of the function f. The data assumes that a single instruction takes 1 µs to execute.

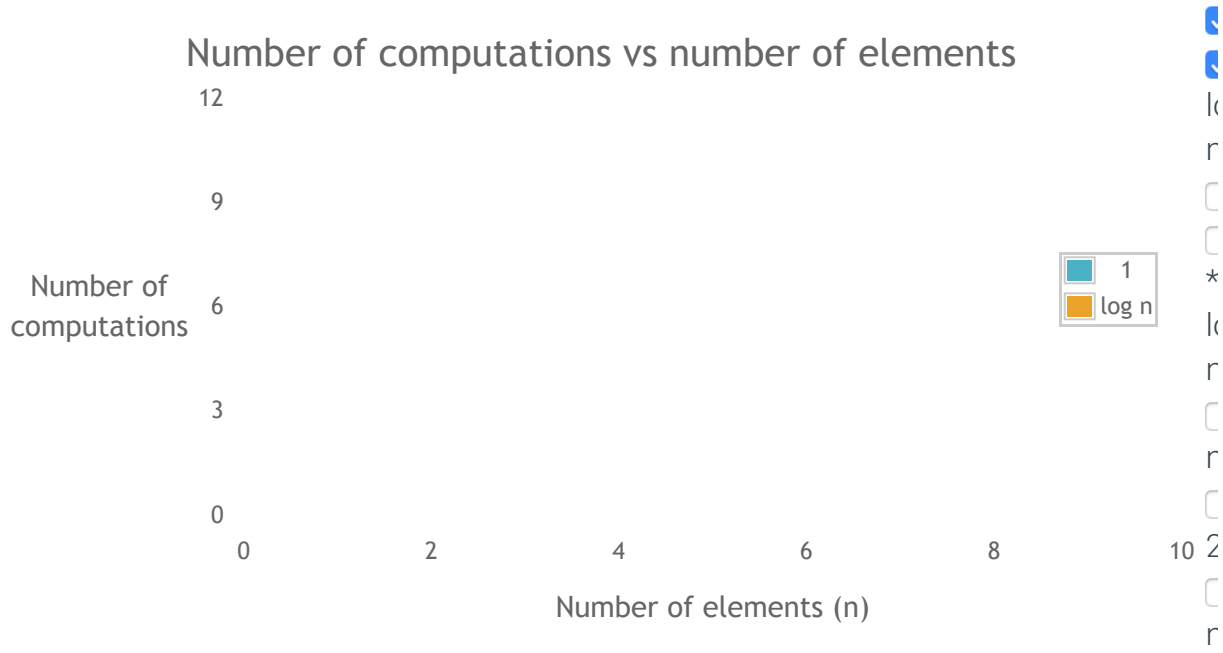
Table 1.10.1: Growth rates for different input sizes.

Function	N = 10	N = 50	N = 100	N = 1000	N = 10000	N = 100000
$log_2 N$	3.3 µs	5.65 µs	6.6 µs	9.9 µs	13.3 µs	16.6 µs
N	10 µs	50 µs	100 µs	1000 µs	10 ms	100 ms
$N log_2 N$	.03 ms	.28 ms	.66 ms	.099 s	.132 s	1.66 s
N^2	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
N^3	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years

2^N	.001 s	35.7 years	> 1000 years

The interactive tool below illustrates graphically the growth rate of commonly encountered functions.

PARTICIPATION ACTIVITY 1.10.4: Computational complexity graphing tool.



Common Big O complexities

Many commonly used algorithms have running time functions that belong to one of a handful of growth functions. These common Big O notations are summarized in the following table. The table shows the Big O notation, the common word used to describe algorithms that belong to that notation, and an example with source code. Clearly, the best algorithm is one that has constant time complexity. Unfortunately, not all problems can be solved using constant complexity algorithms. In fact, in many cases, computer scientists have proven that certain types of problems can only be solved using quadratic or exponential algorithms.

Figure 1.10.2: Runtime complexities for various code examples.

Notation	Name	Example pseudocode
O(1)	Constant	

		`FindMin(x, y) {`
		`  if (x < y) {`
		`  return x`
		`  }`
		`  else {`
		`  return y`
		`  }`
		`}`

O(log N)	Logarithmic	`BinarySearch(numbers, N, key) {`
		`  mid = 0`
		`  low = 0`
		`  high = N - 1`
		
		`  while (high >= low) {`
		`  mid = (high + low) / 2`
		`  if (numbers[mid] < key) {`
		`  low = mid + 1`
		`  }`
		`  else if (numbers[mid] > key) {`
		`  high = mid - 1`
		`  }`
		`  else {`
		`  return mid`
		`  }`
		`  }`
		
		`  return -1  // not found`
		`}`

O(N)	Linear	`LinearSearch(numbers, numbersSize, key) {`
		`  for (i = 0; i < numbersSize; ++i) {`
		`  if (numbers[i] == key) {`
		`  return i`
		`  }`
		`  }`
		
		`  return -1 // not found`
		`}`

O(N log N)	Linearithmic	`MergeSort(numbers, i, k) {`
		`  j = 0`
		`  if (i < k) {`
		`  j = (i + k) / 2  // Find midpoint`
		
		`  MergeSort(numbers, i, j)  // Sort left part`
		`  MergeSort(numbers, j + 1, k) // Sort right part`
		`  Merge(numbers, i, j, k)  // Merge parts`
		`  }`
		`}`

$O(N^2)$	Quadratic	

		`SelectionSort(numbers, numbersSize) {`
		`  for (i = 0; i < numbersSize; ++i) {`
		`  indexSmallest = i`
		`  for (j = i + 1; j < numbersSize; ++j) {`
		`  if (numbers[j] < numbers[indexSmallest]) {`
		`  indexSmallest = j`
		`  }`
		`  }`
		
		`  temp = numbers[i]`
		`  numbers[i] = numbers[indexSmallest]`
		`  numbers[indexSmallest] = temp`
		`}`

$O(c^N)$	Exponential	`Fibonacci(N) {`
		`  if ((1 == N) || (2 == N)) {`
		`  return 1`
		`  }`
		`  return Fibonacci(N-1) + Fibonacci(N-2)`
		`}`

PARTICIPATION ACTIVITY 1.10.5: Big O notation and growth rates.

1) O(5) has a _____ runtime complexity.
   - constant
   - linear
   - exponential

2) O(N log N) has a _____ runtime complexity.
   - constant
   - linearithmic
   - logarithmic

3) $O(N + N^2)$ has a _____ runtime complexity.
   - linear-quadratic
   - exponential
   - quadratic

4) A linear search has a _____ runtime complexity.

- O(log N)
- O(N)
- O(N^2)

5) A selection sort has a _____ runtime complexity.

- O(N)
- O(N log N)
- O(N^2)

1.11 Algorithm analysis

Worst-case algorithm analysis

To analyze how runtime of an algorithm scales as the input size increases, we first determine how many operations the algorithm executes for a specific input size, N. Then, the big-O notation for that function is determined. Algorithm runtime analysis often focuses on the worst-case runtime complexity. The ***worst-case runtime*** of an algorithm is the runtime complexity for an input that results in the longest execution. Other runtime analyses include best-case runtime and average-case runtime. Determining the average-case runtime requires knowledge of the statistical properties of the expected data inputs.

PARTICIPATION ACTIVITY 1.11.1: Runtime analysis: Finding the max value.

Animation captions:

1. Runtime analysis determines the total number of operations. Operations include assignment, addition, comparison, etc.
2. The for loop iterates N times, but the for loop's initial expression i = 0 is executed once.
3. For each loop iteration, the increment and comparison expressions are each executed once. In the worst-case, the if's expression is true, resulting in 2 operations.
4. One additional comparison is made before the loop ends.
5. The function f(N) specifies the number of operations executed for input size N. The big-O notation for the function is the algorithm's worst-case runtime complexity.

PARTICIPATION ACTIVITY 1.11.2: Worst-case runtime analysis.

1) Which function best represents the number of operations in the worst-case?

```
i = 0
sum = 0
while (i < N) {
   sum = sum + numbers[i]
   ++i
}
```

- f(N) = 3N + 2
- f(N) = 3N + 3
- f(N) = 2 + N (N + 1)

2) What is the big-O notation for the worst-case runtime?

```
negCount = 0
for(i = 0; i < N; ++i) {
   if (numbers[i] < 0 ) {
      ++negCount
   }
}
```

- f(N) = 2 + 4N + 1
- O(4N + 3)
- O(N)

3) What is the big-O notation for the worst-case runtime?

```
for (i = 0; i < N; ++i) {
   if ((i % 2) == 0) {
      outVal[i] = inVals[i] * i
   }
}
```

- O(1)
- O($\frac{N}{2}$)
- O(N)

4) What is the big-O notation for the worst-case runtime?

```
nVal = N
steps = 0
while (nVal > 0) {
   nVal = nVal / 2
   steps = steps + 1
}
```

- O(log N)
- O($\frac{N}{2}$)
- O(N)

5) What is the big-O notation for the **best-case** runtime?

```
i = 0
belowMinSum = 0.0
belowMinCount = 0
while (i < N && numbers[i] <=
maxVal) {
   belowMinCount = belowMinCount +
1
   belowMinSum = numbers[i]
   ++i
}
avgBelow = belowMinSum /
belowMinCount
```

- O(1)
- O(N)

Counting constant time operations

For algorithm analysis, the definition of a single operation does not need to be precise. An operation can be any statement (or constant number of statements) that has a constant runtime complexity, O(1). Since constants are omitted in big-O notation, any constant number of constant time operations is O(1). So, precisely counting the number of constant time operations in a finite sequence is not needed. Ex: An algorithm with a single loop that execute 5 operations before the loop, 3 operations each loop iteration, and 6 operations after the loop would have a runtime of f(N) = 5 + 3N + 6, which can be written as O(1) + O(N) + O(1) = O(N). If the number of operations before the loop was 100, the big-O notation for those operation is still O(1).

PARTICIPATION ACTIVITY 1.11.3: Simplified runtime analysis: A constant number of constant time operations is O(1).

Animation captions:

1. Constants are omitted in big-O notation, so any constant number of constant time operations is O(1).

2. The for loop iterates N times. Big-O complexity can be written as a composite function and simplified.

PARTICIPATION ACTIVITY 1.11.4: Constant time operations.

1) A for loop of the form `for (i = 0; i < N; ++i) {}` that does not have nested loops or function calls, and does not modify i in the loop will always have a complexity of O(N).

- True
- False

2) The complexity of the algorithm below is O(1).

```
if (timeHour < 6) {
   tollAmount = 1.55
}
else if (timeHour < 10) {
   tollAmount = 4.65
}
else if (timeHour < 18) {
   tollAmount = 2.35
}
else {
   tollAmount = 1.55
}
```

- True
- False

3) The complexity of the algorithm below is O(1).

```
for (i = 0; i < 24; ++i) {
   if (timeHour < 6) {
      tollSchedule[i] = 1.55
   }
   else if (timeHour < 10) {
      tollSchedule[i] = 4.65
   }
   else if (timeHour < 18) {
      tollSchedule[i] = 2.35
   }
   else {
      tollSchedule[i] = 1.55
   }
}
```


- True

False

Runtime analysis of nested loops

Runtime analysis for nested loops requires summing the runtime of the inner loop over each outer loop iteration. The resulting summation can be simplified to determine the big-O notation.

PARTICIPATION ACTIVITY 1.11.5: Runtime analysis of nested loop: Selection sort algorithm.

Animation captions:

1. For each iteration of the outer loop, the runtime of the inner loop is determined and added together to form a summation. For iteration i = 0, the inner loop executes N - 1 iterations.
2. For i = 1, the inner loop iterates N - 2 times: iterating from j = 2 to N - 1.
3. For i = N - 3, the inner loop iterates twice: iterating from j = N - 2 to N - 1. For i = N - 2, the inner loop iterates once: iterating from j = N - 1 to N - 1.
4. For i = N - 1, the inner loop iterates 0 times. The summation is the sum of a consecutive sequence of numbers from N - 1 to 0.
5. The sequence contains N / 2 pairs, each summing to N - 1, and can be simplified.
6. Each iteration of the loops requires a constant number of operations, which is defined as the constant c.
7. Additionally, each iteration of the outer loop requires a constant number of operations, which is defined as the constant d.
8. Big-O notation omits the constant values, and the runtime is equal to the summation of the total inner loop iterations.

Figure 1.11.1: Common summation: Summation of consecutive numbers.

$$(N-1)+(N-2)+\cdots+2+1=\frac{N(N-1)}{2}=O(N^2)$$

PARTICIPATION ACTIVITY 1.11.6: Nested loops.

Determine the big-O worst-case runtime for each algorithm.

1)

```
for (i = 0; i < N; i++)  {
   for (j = 0; j < N; j++) {
      if (numbers[i] < numbers[j])
{
         ++eqPerms
      }
      else {
         ++neqPerms
      }
   }
}
```


- O(N)
- $O(N^2)$

2)

```
for (i = 0; i < N; i++)  {
   for (j = 0; j < (N - 1); j++) {
      if (numbers[j + 1] <
numbers[j]) {
         temp = numbers[j]
         numbers[j] = numbers[j +
1]
         numbers[j + 1] = temp
      }
   }
}
```

- O(N)
- $O(N^2)$

3)

```
for (i = 0; i < N; i = i + 2)  {
   for (j = 0; j < N; j = j + 2) {
      cVals[i][j] = inVals[i] * j
   }
}
```

- O(N)
- $O(N^2)$

4)

```
for (i = 0; i < N; ++i)  {
   for (j = i; j < N - 1; ++j) {
      cVals[i][j] = inVals[i] * j
   }
}
```

- $O(N^2)$
- $O(N^3)$

5)

```
for (i = 0; i < N; ++i) {
   sum = 0
   for (j = 0; j < N; ++j) {
      for (k = 0; k < N; ++k) {
         sum = sum + aVals[i][k] *
bVals[k][j]
      }

      cVals[i][j] = sum
   }
}
```

- O(N)
- $O(N^2)$
- $O(N^3)$

©zyBooks 02/06/20 14:38 652770
JEFFREY WAN
JHUEN605202Spring2020