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, often called fields, with a name associated with each subitem.
Array	An ***array*** is a data structure that stores an ordered list of items, where 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

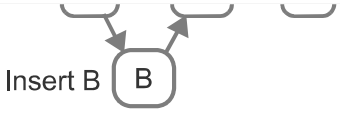
3) A list node's data can store a record with multiple subitems.
   - True
   - False

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 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 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 1.2.1: Computational problems and algorithms.

Computational problem:

Input: Array of integers

13	7	45	-3	92	17

Question: What is the maximum value in the input array?

Output: Maximum integer in input array
92

Algorithm:

```
FindMax(inputArray) {
   max = inputArray[0]

   for (i = 1; i < inputArray.size; ++i) {
      if (inputArray[i] > max) {
         max = inputArray[i];
      }
   }

   return max
}
```


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 1.2.2: 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 1.2.1: 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 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 1.2.3: 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.

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 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 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 1.2.4: Example NP-complete problem: Cliques.

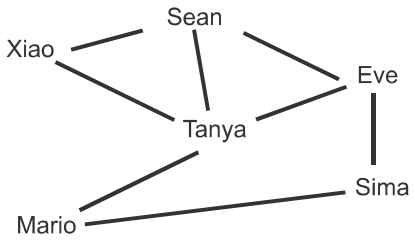
Computational problem:

Input social network:

Input:
G: Graph of social network, where vertices represent individuals and edges represent mutual friendship
K: integer

Question: Does a set of K people who all know each other exist?

Output:
Boolean value



Problem is NP-complete

Does a set of 3 people who all know each other exist? Yes
Does a set of 4 people who all know each other exist? No

Animation content:

Static figure:
Pictured are a computational problem and an input social network.

Computational problem:
Input: "G: Graph of social network, where vertices represent individuals and edges represent mutual friendship Boolean value Problem is NP-complete. K: integer"
Question: Does a set of K people who all know each other exist?
Output: Boolean value

This problem is NP-complete

Six names exist in the graph, some of which are connected to each other with an edge. The following table gives a source-vertex name, followed by a colon, with all the other names connected to the source-vertex name after the colon. (e.g., Xiao is connected to Sean and Tanya)
Xiao: Sean, Tanya
Sean: Xiao, Tanya, Eve
Eve: Sean, Tanya, Sima
Tanya: Xiao, Sean, Eve, Mario
Mario: Tanya, Sima
Sima: Eve, Mario

Two labels exist in the lower-right:
"Does a set of 3 people who all know each other exist? Yes"
"Does a set of 4 people who all know each other exist? No"

Step 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.
The computational problem visible in the static figure appears.

Step 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.
The input social network graph appears along with the question, "Does a set of 3 people who all know each other exist?" The names in the graph Xiao, Sean, Tanya, are highlighted, making note that the three all know each other. The question now is answered with the text "Yes."

Step 3: For K = 4, no set of 4 individuals who all know each other exists, and the algorithm should return no.
Some question and answer text appears: "Does a set of 4 people who all know each other exist? No"

Step 4: This problem is equivalent to the clique decision problem, which is NP-complete, and no known polynomial time algorithm exists.

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 individuals 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

PARTICIPATION ACTIVITY 1.2.5: 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

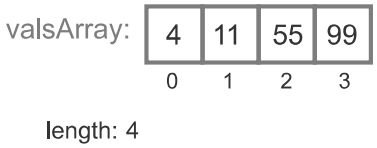
1.3 Relation between data structures and algorithms

Algorithms for data structures

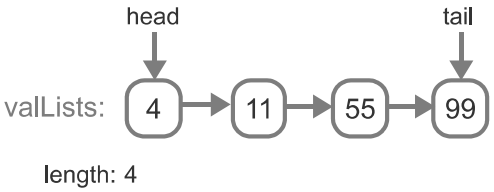
Data structures not only define how data is organized and stored, but also the operations performed on the data structure. While common operations include inserting, removing, and searching for data, the algorithms to implement those operations are typically specific to each data structure. Ex: Appending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

PARTICIPATION ACTIVITY 1.3.1: A list avoids the shifting problem.

Append(valsArray, 99)



Append(valsList, 99)



Algorithm for appending to array

```
ArrayAppend(array, newItem) {
   currentSize = array→length
   Increase array size by one
   array[currentSize] = newItem
}
```

Algorithm for appending to linked-list

```
ListAppend(list, newNode) {
   if (list→head == null) { // List empty
      list→head = newNode
      list→tail = newNode
   }
   else{
      list→tail→next = newNode
      list→tail = newNode
   }
   list→length++
}
```

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 1.3.2: 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 1.3.1: 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 1.3.3: 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 elements with no name and negative sales.

- True
- False

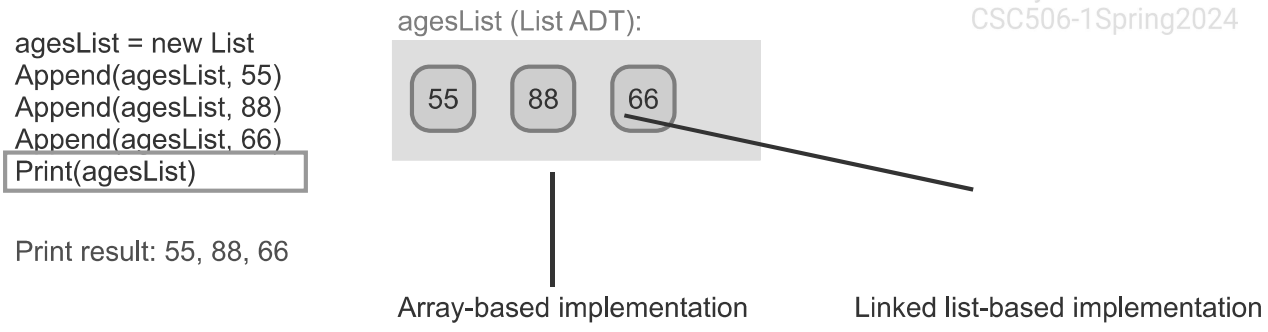
1.4 Abstract data types

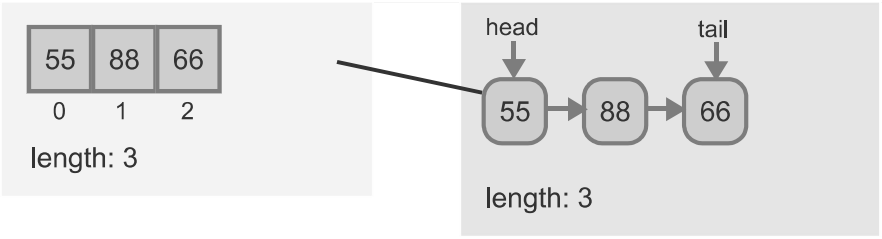
Abstract data types (ADTs)

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 1.4.1: 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 1.4.2: 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 1.4.1: 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 1.4.3: 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.

If unable to drag and drop, refresh the page.

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

	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

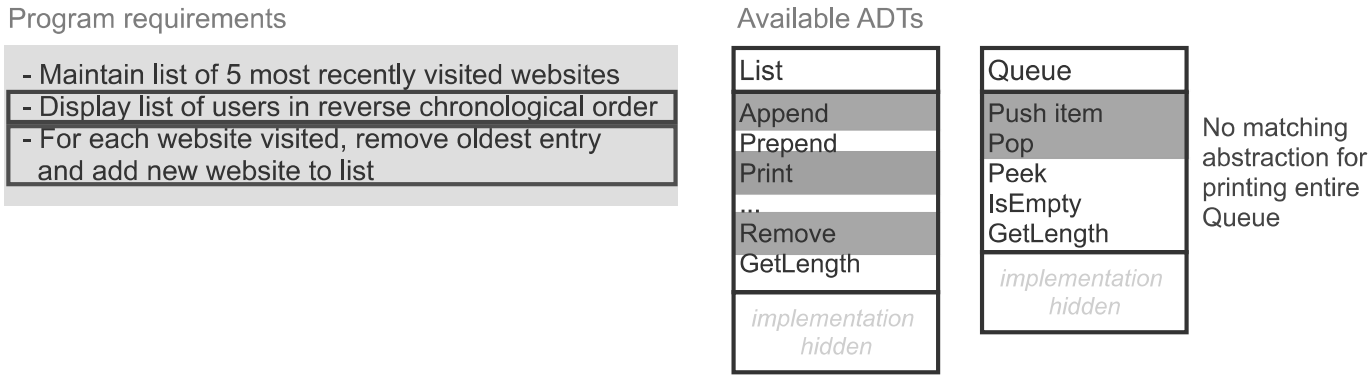
1.5 Applications of ADTs

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 1.5.1: 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 1.5.2: Programming with ADTs.

Consider the example in the animation above.

1) The _____ ADT is the better match for the program's requirements.
   - ○ queue
   - ○ list

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 1.5.1: 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 1.5.3: 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.

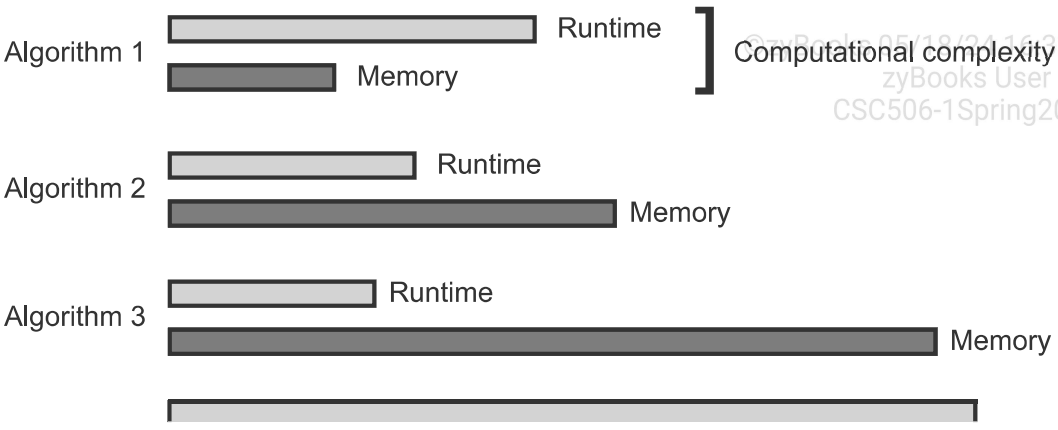
- True
- False

1.6 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 1.6.1: 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 1.6.2: 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 1.6.3: Linear search best and worst cases.

```
LinearSearch(numbers, numbersSize, key) {
  i = 0
  while (i < numbersSize) {
     if (numbers[i] == key)
        return i
     i = i + 1
  }

  return -1 // not found
}
```

numbers:

54	79	26	91	29	33

key = 26: neither best nor worst case

key = 54: best case

key = 82: worst case

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 1.6.4: 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.

Worst case | **Neither best nor worst case** | **Best 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 1.6.5: 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 1.6.6: FindMax space complexity and auxiliary space complexity.

```
FindMax(list, listSize) {
   if (listSize >= 1) {
      maximum = list[0]
      i = 1
      while (i < listSize) {
         if (list[i] > maximum) {
            maximum = list[i]
         }
         i = i + 1
      }
      return maximum
   }
}
```

Space complexity: S(N) = N + 3

Auxiliary space complexity: S(N) = 2

Memory (input)

listSize: fixed-size integer

list: Array of N integers

Memory (non-input)

maximum: fixed-size integer

i: fixed-size integer

2 fixed-size variables

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 1.6.7: 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?

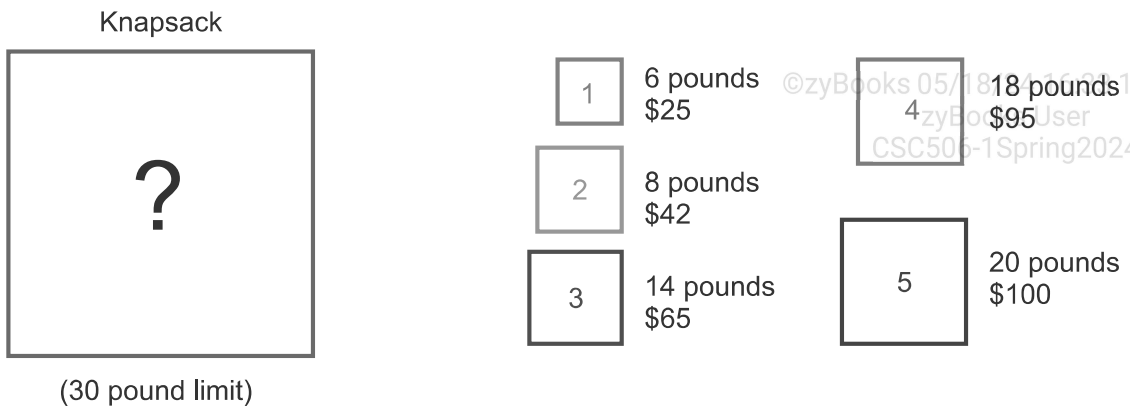
   - S(N) = N + k
   - S(N) = k

1.7 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 1.7.1: Introduction to the knapsack problem.



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 1.7.2: 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 1.7.3: 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.

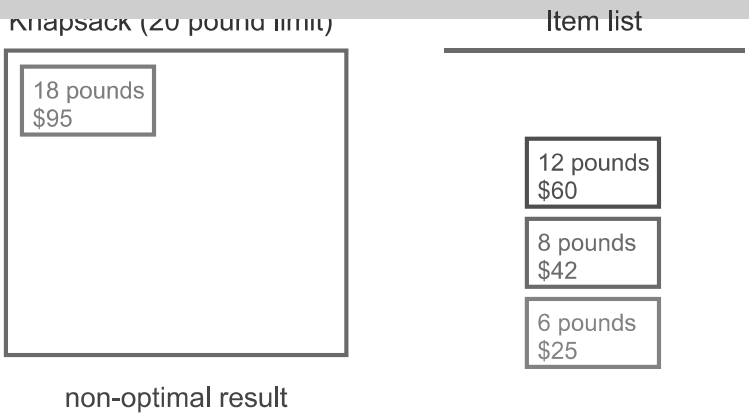
Heuristic optimization

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 1.7.4: Non-optimal, heuristic algorithm to solve the 0-1 knapsack.

```
Knapsack01(knapsack, itemList, itemListSize) {
   Sort itemList descending by value
   remaining = knapsack→maximumWeight
   for (i = 0; i < itemListSize; i++) {
      if (itemList[i]→weight <= remaining) {
         Put itemList[i] in knapsack
         remaining = remaining - itemList[i]→weight
      }
   }
}
```



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 1.7.5: 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.

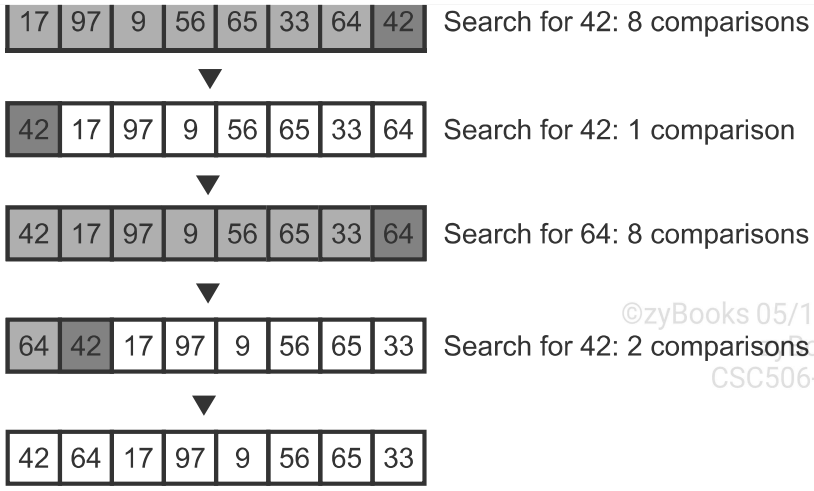
Self-adjusting heuristic

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 1.7.6: Move-to-front self-adjusting heuristic.

Original list:



Frequent searches for already-searched-for keys lowers the total number of comparisons

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 1.7.7: 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.

1.8 Python: Heuristics

0-1 Knapsack problem heuristic

The general knapsack problem seeks to maximize the total value of items placed into a knapsack such that the total weight of items in the knapsack doesn't exceed a predetermined weight. The 0-1 knapsack problem imposes the restriction that each item can be added at most once. A heuristic algorithm to solve the knapsack problem first sorts items in descending order by value, and then iteratively places the most valuable items that fit within the remaining space into the knapsack until no more items can be added.

To construct a Python program to solve the knapsack problem using a heuristic, two classes are defined:

- An Item class to store each item's weight and value
- A Knapsack class to store the knapsack's maximum predetermined weight and the list of items the knapsack will hold

The heuristic is defined in the knapsack_01() function. To sort the original list of items by value, the operator module's attrgetter() function (attribute getter) is imported and used in list's sort() method to identify the 'value' attribute as the sorting key. The `reverse = True` argument passed to list's sort method() sorts the items in descending order. Then the first item of the sorted list is added to the knapsack, changing the knapsack's remaining weight. The algorithm continues until the knapsack is full or until the next item in the list weighs more than the remaining weight in the knapsack.

Figure 1.8.1: 0-1 Knapsack problem heuristic.

```
from operator import attrgetter

class Item:
    def __init__(self, item_weight, item_value):
        self.weight = item_weight
        self.value = item_value


class Knapsack:
    def __init__(self, weight, items):
        self.max_weight = weight
        self.item_list = items


def knapsack_01(knapsack, item_list):
    # Sort the items in descending order based on value
    item_list.sort(key = attrgetter('value'), reverse = True)

    remaining = knapsack.max_weight
    for item in item_list:
        if item.weight <= remaining:
            knapsack.item_list.append(item)
            remaining = remaining - item.weight


# Main program
item_1 = Item(6, 25)
item_2 = Item(8, 42)
item_3 = Item(12, 60)
item_4 = Item(18, 95)
item_list = [item_1, item_2, item_3, item_4]
initial_knapsack_list = []

max_weight = int(input('Enter maximum weight the knapsack can hold: '))

knapsack = Knapsack(max_weight, initial_knapsack_list)
knapsack_01(knapsack, item_list)

print ('Objects in knapsack')
i = 1
sum_weight = 0
sum_value = 0

for item in knapsack.item_list:
    sum_weight += item.weight
    sum_value += item.value
    print ('%d: weight %d, value %d' % (i, item.weight,item.value))
    i += 1
print()

print('Total weight of items in knapsack: %d' % sum_weight)
print('Total value of items in knapsack: %d' % sum_value)
```



```
Enter maximum weight the knapsack can hold: 40
Objects in knapsack
1: weight 18, value 95
2: weight 12, value 60
3: weight 8, value 42

Total weight of items in knapsack: 38
Total value of items in knapsack: 197
```

©zyBooks 05/18/24 16:33 1870315
zyBooks User
CSC506-1Spring2024

PARTICIPATION ACTIVITY 1.8.1: 0-1 Knapsack heuristic.

For questions 1 and 2, refer to the code above and assume `max_weight = 30`.

1) How many items are put into the knapsack using the heuristic?

- 1
- 2
- 3

2) Does the heuristic find the optimal solution to the 0-1 knapsack problem?

- Yes
- No

3) Using the heuristic, which items are added to the knapsack assuming:

```
item_1 = Item(6, 25)
item_2 = Item(8, 42)
item_3 = Item(12, 60)
item_4 = Item(18, 95)

max_weight= 28
```

- item_2 and item_1
- item_4, item_3, and item_2
- item_4 and item_2

4) Using the heuristic, what is the weight sum of the items added to the knapsack assuming:

```
item_1 = Item(6, 25)
item_2 = Item(8, 42)
item_3 = Item(12, 60)
item_4 = Item(18, 95)

max_weight= 32
```

- 30
- 32

zyDE 1.8.1: 0-1 Knapsack problem heuristic.

The program below employs the 0-1 knapsack heuristic to add three items (item_4, item_3, item_2) to the knapsack; the sum of those items

(38) is less than knapsack's max_weight (40). You can try running the program with different maximum weights to see how many items are added.

main.py Load default template...

```
from operator import attrgetter

class Item:
    def __init__(self, item_weight, item_value):
        self.weight = item_weight
        self.value = item_value


class Knapsack:
    def __init__(self, weight, items):
        self.max_weight = weight
        self.item_list = items


def knapsack_01(knapsack, item_list):
    # Sort the items in descending order based on value
    item_list.sort(key = attrgetter('value'), reverse =

```


40

Run

1.9 Greedy algorithms

Greedy algorithm

A ***greedy algorithm*** is an algorithm that, when presented with a list of options, chooses the option that is optimal at that point in time. The choice of option does not consider additional subsequent options, and may or may not lead to an optimal solution.

PARTICIPATION ACTIVITY 1.9.1: MakeChange greedy algorithm.

```
MakeChange(amount) {
   while (amount >= 25) {
      Add quarter
      amount = amount - 25
   }
   while (amount >= 10) {
      Add dime
      amount = amount - 10
   }
   while (amount >= 5) {
      Add nickel
      amount = amount - 5
   }
   while (amount >= 1) {
      Add penny
      amount = amount - 1
   }
}
```

MakeChange(91)

3 quarters
1 dime
1 nickel
1 penny

} 91 cents, minimal number of coins

Animation captions:

1. The change making algorithm uses quarters, dimes, nickels, and pennies to make change equaling the specified amount.
2. The algorithm chooses quarters as the optimal coins, as long as the remaining amount is >= 25.
3. Dimes offer the next largest amount per coin, and are chosen while the amount is >= 10.
4. Nickels are chosen next. The algorithm is greedy because the largest coin <= the amount is always chosen.
5. Adding one penny makes 91 cents.
6. This greedy algorithm is optimal and minimizes the total number of coins, although not all greedy algorithms are optimal.

PARTICIPATION ACTIVITY 1.9.2: Greedy algorithms.

1) If the MakeChange function were to make change for 101, what would be the result?

- 101 pennies
- 4 quarters and 1 penny
- 3 quarters, 2 dimes, 1 nickel, and 1 penny

2) A greedy algorithm is attempting to minimize costs and has a choice between two items with equivalent functionality: the first costing $5 and the second costing $7. Which will be chosen?

- The $5 item
- The $7 item
- The algorithm needs more information to choose

3) A greedy algorithm always finds an optimal solution.

- True
- False

Fractional knapsack problem

The ***fractional knapsack problem*** is the knapsack problem with the potential to take each item a fractional number of times, provided the fraction is in the range [0.0, 1.0]. Ex: A 4 pound, $10 item could be taken 0.5 times to fill a knapsack with a 2 pound weight limit. The resulting knapsack would be worth $5.

While a greedy solution to the 0-1 knapsack problem is not necessarily optimal, a greedy solution to the fractional knapsack problem is optimal. First, items are sorted in descending order based on the value-to-weight ratio. Next, one of each item is taken from the item list, in order, until taking 1 of the next item would exceed the weight limit. Then a fraction of the next item in the list is taken to fill the remaining weight.

Figure 1.9.1: FractionalKnapsack algorithm.

```
FractionalKnapsack(knapsack, itemList, itemListSize) {
   Sort itemList descending by item's (value / weight)
ratio
   remaining = knapsack→maximumWeight
   for each item in itemList {
      if (item→weight <= remaining) {
         Put item in knapsack
         remaining = remaining - item→weight
      }
      else {
         fraction = remaining / item→weight
         Put (fraction * item) in knapsack
         break
      }
   }
}
```


PARTICIPATION ACTIVITY 1.9.3: Fractional knapsack problem.

Suppose the following items are available: 40 pounds worth $80, 12 pounds worth $18, and 8 pounds worth $8.

1) Which item has the highest value-to-weight ratio?
   - 40 pounds worth $80
   - 12 pounds worth $18
   - 8 pounds worth $8

2) What would FractionalKnapsack put in a 20-pound knapsack?
   - One 12-pound item and one 8-pound item
   - One 40-pound item
   - Half of a 40-pound item

3) What would FractionalKnapsack put in a 48-pound knapsack?
   - One 40-pound item and one 8-pound item
   - One 40-pound item and 2/3 of a 12-pound item

Activity selection problem

The ***activity selection problem*** is a problem where 1 or more activities are available, each with a start and finish time, and the goal is to build the largest possible set of activities without time conflicts. Ex: When on vacation, various activities such as museum tours or mountain hikes may be available. Since vacation time is limited, the desire is often to engage in the maximum possible number of activities per day.

A greedy algorithm provides the optimal solution to the activity selection problem. First, an empty set of chosen activities is allocated. Activities are then sorted in ascending order by finish time. The first activity in the sorted list is marked as the current activity and added to the set of chosen activities. The algorithm then iterates through all activities after the first, looking for a next activity that starts after the current activity ends. When such a next activity is found, the next activity is added to the set of chosen activities, and the next activity is reassigned as the current. After iterating through all activities, the chosen set of activities contains the maximum possible number of non-conflicting activities from the activities list.

PARTICIPATION ACTIVITY 1.9.4: Activity selection problem algorithm.

```
ActivitySelection(activities, activitiesSize) {
   chosen = Allocate new, empty activity set
   Sort activities ascending by finish time
   current = activities[0]
   Add current to chosen
   for (i = 1; i < activitiesSize; i++) {
      if (activities[i]→start >= current→finish) {
         Add activities[i] to chosen
         current = activities[i]
      }
   }
   return chosen
}
```

activities

0	{ "History museum tour", 9 AM, 10 AM }
1	{ "Morning mountain hike", 9 AM, 12 PM }
2	{ "Boat tour", 11 AM, 2 PM }
3	{ "Day mountain hike", 1 PM, 4 PM }
4	{ "Hang gliding", 2 PM, 4 PM }
5	{ "Snorkeling", 3 PM, 5 PM }
6	{ "Fireworks show", 8 PM, 9 PM }
7	{ "Night movie", 7 PM, 9 PM }

chosen

{ "History museum tour", 9 AM, 10 AM }
{ "Boat tour", 11 AM, 2 PM }
{ "Hang gliding", 2 PM, 4 PM }
{ "Fireworks show", 8 PM, 9 PM }

Optimal schedule, in terms of maximizing number of activities

Animation captions:

1. Activities are first sorted in ascending order by finish time. The set of chosen activities initially has the activity that finishes first.
2. The morning mountain hike does not start after the history museum tour finishes and is not added to the chosen set of activities.
3. The boat tour is the first activity to start after the history museum tour finishes, and is the "greedy" choice.
4. Hang gliding and the fireworks show are chosen as 2 additional activities.
5. The maximum possible number of non-conflicting activities is 4, and 4 have been chosen.

PARTICIPATION ACTIVITY 1.9.5: ActivitySelection algorithm.

1) The fireworks show and the night movie both finish at 9 PM, so the sorting algorithm could have swapped the order of the 2. If the 2 were swapped, the number of chosen activities would not be affected.
   - ○ True
   - ○ False

2) Changing snorkeling's ____ would cause snorkeling to be added to the chosen activities.
   - ○ start time from 3 PM to 4 PM
   - ○ finish time from 5 PM to 4 PM

3) Regardless of any changes to the activity list, the activity with the ____ will always be in the result.
   - ○ earliest start time
   - ○ earliest finish time
   - ○ longest length

1.10 Python: Greedy algorithms

Greedy algorithm

A greedy algorithm solves a problem by assuming that the optimal choice *at a given moment* during the algorithm will also be the optimal choice overall. Greedy algorithms tend to be efficient, but certain types of problems exist where greedy algorithms don't find the *best* or *optimal* solution. However, greedy algorithms produce both efficient and optimal solutions for many problems, including the fractional knapsack problem and the activity selection problem.

Fractional knapsack problem

The fractional knapsack problem is similar to the regular knapsack problem, except that fractional pieces of any given item are allowed to be selected. The optimal solution can be achieved by first sorting the items by their value-to-weight ratio, in descending order. The items are then added to the knapsack in that order, taking whole units of items until only a fraction of the next item is possible. The highest fraction of that item that will fit in the knapsack is taken, and the algorithm ends.

The list's sort() method is used with a custom key function, value_to_weight_ratio(). The key function tells the sort() method what value to use to sort each item. For solving fractional knapsack problem, sort() will use the item's value divided by the item's weight. The items are sorted in descending order, so the call to sort() passes the `reverse = True` parameter.

Item objects have a `fraction` data member that is assigned with the fraction of the item (in the range [0.0, 1.0]) in the Knapsack. Items are assumed to be added fully, so fraction is assigned with 1.0 when the Item object is constructed; the value of the fraction data member is modified in the fractional_knapsack() function when only a part of the item fits in the knapsack.

Figure 1.10.1: Fractional knapsack greedy algorithm.

```
# An individual item with a weight and value
class Item:
    def __init__(self, item_weight, item_value):
        self.weight = item_weight
        self.value = item_value
        self.fraction = 1.0

# The knapsack that contains a list of items and a maximum
# total weight
class Knapsack:
    def __init__(self, weight, items):
        self.max_weight = weight
        self.item_list = items

# A key function to be used to sort the items
def value_to_weight_ratio(item):
    return item.value / item.weight

# The Fractional Knapsack algorithm.
def fractional_knapsack(knapsack, item_list):
    # Sort the items in descending order based on value/weight
    item_list.sort(key = value_to_weight_ratio, reverse = True)

    remaining = knapsack.max_weight
    for item in item_list:
        # Check if the full item can fit into the knapsack or
        # only a fraction
        if item.weight <= remaining:
            # The full item will fit.
            knapsack.item_list.append(item)
            remaining = remaining - item.weight

        else:
            # Only a fractional part of the item will fit. Add
that
            # fraction of the item, and then exit.
            item.fraction = remaining / item.weight
            knapsack.item_list.append(item)
            break
```


To test the algorithm, a Knapsack object is created with a user-defined capacity, along with a list of available items. The fractional_knapsack() method is executed. Then, the items in the knapsack are displayed with the items' weight, cost, and fraction.

Figure 1.10.2: A program to test the fractional knapsack greedy algorithm. The knapsack is given a maximum weight of 35.

```
# Main program to test the fractional knapsack algorithm.
item_1 = Item(6, 25)
item_2 = Item(8, 42)
item_3 = Item(12, 60)
item_4 = Item(18, 95)
item_list = [item_1, item_2, item_3, item_4]
initial_knapsack_list = []

# Ask the user for the knapsack's maximum capacity.
max_weight = int(input('Enter maximum weight the knapsack can hold: '))


# Construct the knapsack object, then run the fractional_knapsack
# algorithm on it.
knapsack = Knapsack(max_weight, initial_knapsack_list)
fractional_knapsack(knapsack, item_list)

# Output the information about the knapsack. Show the contents
# of the knapsack, and the fraction of each item.
print ('Objects in knapsack')
i = 1
sum_weight = 0
sum_value = 0
for item in knapsack.item_list:
    sum_weight += item.weight * item.fraction
    sum_value += item.value * item.fraction
    print ('%d: %0.1f of weight %0.1f, value %0.1f' %
           (i, item.fraction, item.weight, item.value * item.fraction))
    i += 1
print()

# Display the total weight of the items as well as the total value.
print('Total weight of items in knapsack: %d' % sum_weight)
print('Total value of items in knapsack: %d' % sum_value)
```



```
Enter maximum weight the knapsack can hold: 35
Objects in knapsack
1: 1.0 of weight 18.0, value 95.0
2: 1.0 of weight 8.0, value 42.0
3: 0.8 of weight 12.0, value 45.0

Total weight of items in knapsack: 35.0
Total value of items in knapsack: 182.0
```

PARTICIPATION ACTIVITY 1.10.1: The fractional knapsack algorithm.

1) The items are first sorted in descending order by weight.

   - True
   - False

2) The algorithm finds the largest number of items that can fit into the knapsack.

- True
- False

3) The algorithm always fills the knapsack to the maximum weight (so long as the list has enough items to do so).

- True
- False

zyDE 1.10.1: Fractional knapsack greedy algorithm.

Try giving the knapsack different maximum weights. Note that all items in the knapsack have their full value (fraction = 1.0), but the final item has a fraction between 0.0 and 1.0 (inclusive). A maximum weight of 38 is a special case. What is unusual about the results? Are the results still technically correct? How could you modify the program to make the results more natural looking?

©zyBooks 05/18/24 16:33 1870315
zyBooks User
CSC506-1Spring2024

main.py Load default template...

```
# An individual item with a weight and value
class Item:
    def __init__(self, item_weight, item_value):
        self.weight = item_weight
        self.value = item_value
        self.fraction = 1.0

# The knapsack that contains a list of items and a maxim
# total weight
class Knapsack:
    def __init__(self, weight, items):
        self.max_weight = weight
        self.item_list = items

# A key function to be used to sort the items
def value_to_weight_ratio(item):
    return item.value / item.weight

```


35

Run

Activity selection problem

The activity selection problem defines a set of "activities" (Ex: Tourist activities on a vacation), as well as when these activities start and finish (Ex: Time of day). The optimal solution will schedule the *most* activities possible without having any time conflicts.

The greedy algorithm starts by sorting the activities, using the activity finish times as the sorting key, from earliest to latest. The first activity in the list is selected and added to the set of chosen activities. The second activity in the sorted list is selected only if the activity does not conflict with the first activity. If the second activity *does* conflict with the first, then the activity is not selected. The third activity in the sorted list is selected only if the activity does not conflict with the second activity. The process continues until the entire sorted list of activities is processed.

An activity is represented with a Python class called Activity. Three data members are defined: `name`, `start_time`, and `finish_time`. A `conflicts_with()` method is also defined to determine whether or not a time conflict exists between two Activity objects.

Figure 1.10.3: Activity class for the activity selection problem.

```
class Activity:
    def __init__(self, name, initial_start_time, initial_finish_time):
        self.name = name
        self.start_time = initial_start_time
        self.finish_time = initial_finish_time

    def conflicts_with(self, other_activity):
        # No conflict exists if this activity's finish_time comes
        # at or before the other activity's start_time
        if self.finish_time <= other_activity.start_time:
            return False

        # No conflict exists if the other activity's finish_time
        # comes at or before this activity's start_time
        elif other_activity.finish_time <= self.start_time:
            return False

        # In all other cases the two activity's conflict with each
        # other
        else:
            return True

# Main program to test Activity objects
activity_1 = Activity('History museum tour', 9, 10)
activity_2 = Activity('Morning mountain hike', 9, 12)
activity_3 = Activity('Boat tour', 11, 14)

print('History museum tour conflicts with Morning mountain hike:',
      activity_1.conflicts_with(activity_2))
print('History museum tour conflicts with Boat tour:',
      activity_1.conflicts_with(activity_3))
print('Morning mountain hike conflicts with Boat tour:',
      activity_2.conflicts_with(activity_3))
```

```
History museum tour conflicts with Morning mountain hike: True
History museum tour conflicts with Boat tour: False
Morning mountain hike conflicts with Boat tour: True
```


The `activity_selection()` function takes a list of Activity objects as a parameter and finds an optimal selection of activities using the greedy algorithm.

The algorithm uses the list's sort() method, using the key function `attrgetter` imported from the operator module. This key function allows the sort() method to sort the list based on the indicated item attribute, namely the finish_time attribute.

Figure 1.10.4: activity_selection() function.

```
from operator import attrgetter

def activity_selection(activities):

    # Start with an empty list of selected activities
    chosen_activities = []

    # Sort the list of activities in increasing order of finish_time
    activities.sort(key = attrgetter("finish_time"))

    # Select the first activity, and add it to the chosen_activities list
    current = activities[0]
    chosen_activities.append(current)

    # Process all the remaining activities, in order
    for i in range(1, len(activities)):

        # If the next activity does not conflict with
        # the most recently selected activity, select the
        # next activity
        if not activities[i].conflicts_with(current):
            chosen_activities.append(activities[i])
            current = activities[i]

    # The chosen_activities list is an optimal list of
    # activities with no conflicts
    return chosen_activities

# Program to test Activity Selection greedy algorithm. The
# start_time and finish_time are represented with integers
# (ex. "20" is 20:00, or 8:00 PM).
activity_1 = Activity('Fireworks show', 20, 21)
activity_2 = Activity('Morning mountain hike', 9, 12)
activity_3 = Activity('History museum tour', 9, 10)
activity_4 = Activity('Day mountain hike', 13, 16)
activity_5 = Activity('Night movie', 19, 21)
activity_6 = Activity('Snorkeling', 15, 17)
activity_7 = Activity('Hang gliding', 14, 16)
activity_8 = Activity('Boat tour', 11, 14)

activities = [ activity_1, activity_2, activity_3, activity_4,
               activity_5, activity_6, activity_7, activity_8 ]

# Use the activity_selection() method to get a list of optimal
# activities with no conflicts.
itinerary = activity_selection(activities)
for activity in itinerary:
    # Output the activity's information.
    print('%s - start time: %d, finish time: %d' %
          (activity.name, activity.start_time, activity.finish_time))
```

```
History museum tour - start time: 9, finish time: 10
Boat tour - start time: 11, finish time: 14
Hang gliding - start time: 14, finish time: 16
Fireworks show - start time: 20, finish time: 21
```


©zyBooks 05/18/24 16:33 1870315
zyBooks User
CSC506-1Spring2024

PARTICIPATION ACTIVITY 1.10.2: Activity selection algorithm.

1) The Activity class data members `start_time` and `finish_time` can be assigned with any data type that supports the ≤ operator.

   - True
   - False

2) The optimal result for activity selection is the total amount of time spent in the activities.

   - True
   - False

3) The greedy algorithm for activity selection won't necessarily find the optimal result, but will find a result that is close to optimal.

   - True
   - False

PARTICIPATION ACTIVITY 1.10.3: Testing the activity selection greedy algorithm.

Try running the activity selection greedy algorithm with different activity start and finish times. Ex: What happens if the start time for "Snorkeling" is changed to 16? Can you change the integer values for start and finish times to real datetime.time objects? Does the algorithm still work as expected? (Hint: remember to import the datetime module, and then use `datetime.time(hour=9)` to represent 9AM or `datetime.time(hour=17)` to represent 5PM.)

main.py — Load default template...

```
from operator import attrgetter

class Activity:
    def __init__(self, name, initial_start_time, initial_finish_time):
        self.name = name
        self.start_time = initial_start_time
        self.finish_time = initial_finish_time

    def conflicts_with(self, other_activity):
        # No conflict exists if this activity's finish_time comes
        # at or before the other activity's start_time.
        if self.finish_time <= other_activity.start_time:
            return False
```

```
        # No conflict exists if the other activity's finish_time
        # comes at or before this activity's start_time.
        elif other_activity.finish_time <= self.start_time:
            return False
```

Run

1.11 Greedy Algorithm: Creating A Meal Plan

Specification

In this lab you will write a greedy algorithm that creates a daily food plan according to some restrictions:

- The total calories must not exceed 2000.
- The plan must come as close as possible to matching a target amount of calories from one of the three macronutrients: protein, carbohydrates, or fat.

For example, the user might choose to set a goal of 30% protein for the daily meal plan. Your program will attempt to produce a meal plan that has a total of 2000 calories, with 30% of those calories coming from protein.

Supporting classes

`Food` class

A `Food` object represents a single serving of a food item that can be part of a meal plan.

Data members:

- `name` - a name to identify the food, for output display
- `protein` - grams of protein in one serving
- `carbs` - grams of carbohydrates in one serving
- `fat` - grams of fat in one serving
- `calories` - total calories in one serving (note that calories come from more than just protein, carbs and fat, so this number is greater than the sum of the three nutrients used in this lab)
- `fraction` - the fraction of a serving to be used in a meal plan (between 0.0 and 1.0, inclusive)

- `protein_calories`
- `carbs_calories`
- `fat_calories`

Methods:

- `set_fraction(p)` - assigns the `fraction` data member with `p` (between 0.0 and 1.0 inclusive), and calculates new values for `calories`, `protein_calories`, `carbs_calories` and `fat_calories` accordingly.
- `__str__()` - returns a string representing the food and nutrient contents.

`MealPlan` class

A `MealPlan` object is the collection of `Food` objects selected for a single day's meal plan.

Data members:

- `foods` - the list of `Food` objects that are part of the meal plan
- `total_calories` - total calories of the meal plan
- `total_protein_calories` - total calories from protein in the meal plan
- `total_carbs_calories` - total calories from carbohydrates in the meal plan
- `total_fat_calories` - total calories from fat in the meal plan

Provided methods:

- `percent_nutrient(nutrient)` - returns the current percent value (in range [0.0, 1.0]) of the given nutrient in the current meal plan, by calories.
- `percent_nutrient_with_food(food, nutrient)` - returns the total percent (in range [0.0, 1.0]) of the given nutrient *if* the given food were added. The food item is *not* added to the the meal plan by the method; the method is just for speculation only.
- `calories_with_food(food)` - returns the total calories the meal plan *would have*, if the given food were added. The food item is *not* added to the meal plan by this method; the method is just for speculation only.
- `__str__()` - returns a string listing the foods selected for the meal plan along with summary information about calorie content.

Methods for you to complete:

- `fraction_to_fit_calorie_limit(food, calorie_limit)` - returns the fraction (in the range [0.0, 1.0]) of a serving of the specified food item that is needed to make the entire meal plan exactly match the given calorie limit. If an entire serving can fit and the meal plan will be below the calorie limit, then the method returns 1.0. The food item is *not* added to the meal plan by this method; the method is just for speculation only.
- `fraction_to_fit_nutrient_goal(food, nutrient, goal)` - returns the fraction (in the range [0.0, 1.0]) of a serving of the specified food item that is needed to make the entire meal

plan exactly match the given nutrient composition goal. If an entire serving can fit and the meal plan will be below the nutrient composition goal, then the method returns 1.0. The food item is *not* added to the meal plan by this method; the method is just for speculation only.
- `meets_calorie_limit(calorie_limit, threshold)` - returns `True` if the current meal plan has a total calorie content that falls within the given threshold of the specified calorie limit, otherwise returns `False`.
- `meets_nutrient_goal(nutrient, goal, threshold)` - returns `True` if the current meal plan has a total percent nutrient composition (by calories) that falls within the given threshold of the specified composition goal, otherwise returns `False`.

Be sure to read the description of each method carefully, and test them individually with various values to be sure they work correctly. The main algorithm will depend on these methods working properly!

The main program

When the program is executed, the user is prompted for some information:

- the name of the food data file (a string entered from an input prompt)
- the nutrient (protein, carbs or fat) that the user wishes to set a goal for (selected from a menu)
- the goal amount (percent of calories) the selected nutrient must compose of the meal plan (a float value entered from an input prompt)

Example: selecting a meal plan with 30% protein from the `nutrients_1000.txt` data file:

```
Enter name of food data file: nutrients_1000.txt

        1 - Set maximum protein
        2 - Set maximum carbs
        3 - Set maximum fat
        4 - Exit program

Enter choice (1-4): 1
What percentage of calories from protein is the goal? 30
```

The program must be able to handle errors when entering the nutrient and nutrient goal information; when the user enters a value that is of the wrong type (like a string instead of an integer) or a value that is out of range, then an error message is displayed and the user is prompted again.

Once the information has been input from the user, the meal plan is created. You are to complete the function `create_meal_plan()`, which is called from the main program. The basic algorithm for building the meal plan is:

- Sort the list of available foods according to the *heuristic* described below.
- For each food item in the list:

    - add the food if:
        - it is not "empty" (too few calories or too few of the target nutrient), and
        - an entire serving can be added, or
        - an entire serving can't be added, but there is a fraction of a serving that doesn't violate either constraint (total calories or percent nutrient)
    - if the food item was added, check if the meal plan meets both the target calorie content and the percent nutrient content (within acceptable thresholds, as described below)
        - if the meal plan meets the requirements, exit the loop; otherwise continue to the next food item
- display the final meal plan

Details

- A meal plan matches the calorie limit if total calories fall in the range [1999.9, 2000.1].
- A meal plan matches the nutrient goal if the total percent calories of the target nutrient comes within 0.1%. Ex. If the goal is 30% protein, a protein calorie content in the range [29.9%, 30.1%] is successful.
- Only allow fractional servings greater than or equal to 0.05.
- Only include `Food` objects with a calorie content greater than or equal to 0.1.
- Only include `Food` objects that are greater than or equal to 0.1% calories by the selected nutrient. Ex. If the selected nutrient is protein, ignore `Food` items where `protein_calories / calories < 0.001`.

Greedy algorithm heuristic

Iterate through the food item list in *descending* order of the user's selected nutrient percent composition. Ex. for setting maximum protein content, sort the `Food` object list by decreasing `protein_calories`.

Food data file

Food item information is in a text file with one food item specified per line. A food item line is formatted as follows:

- A text string representing the name of the food item, along with serving size information, followed by a colon (":");
- A float value representing the amount of protein in a single serving, in grams, followed by a comma (",");
- A float value representing the amount of carbohydrates in a single serving, in grams, followed by a comma (",");
- A float value representing the amount of fat in a single serving, in grams, followed by a comma (",");
- A float value representing the total calories in a single serving.

```
<Food Info>: <grams_protein>, <grams_carbs>, <grams_fat>,
<total_calories>
```

The food items do not appear in any particular order. You must complete the `load_food_data()` function to parse the data file line-by-line, and return a list of `Food` objects created from the parsed data.

Examples

1. 30% protein

```
Enter name of food data file: nutrients_1000.txt

        1 - Set maximum protein
        2 - Set maximum carbs
        3 - Set maximum fat
        4 - Exit program

Enter choice (1-4): 1
What percentage of calories from protein is the goal? 30
1: [1.0000] Beef, brisket, flat half, separable lean and fat, trimmed
to 1/8" fat, choice, raw (1 steak / 434.0 grams)
(P=78.6408,C=0.5208,F=96.131,E=1206.52)
2: [0.3207] Turkey, retail parts, drumstick, meat and skin, cooked,
roasted (1 drumstick / 275.0 grams)
(P=77.5775,C=0.0,F=25.7675,E=173.7324451060661)
3: [1.0000] Fast foods, submarine sandwich, turkey, roast beef and ham
on white bread with lettuce and tomato (12 inch sub / 413.0 grams)
(P=44.0258,C=84.0868,F=9.9946,E=602.98)
4: [0.0502] Cheese, mozzarella, nonfat (1 cup, shredded / 113.0 grams)
(P=35.821,C=3.955,F=0.0,E=7.994115976331346)
5: [0.1840] Gravy, au jus, canned (1 can / 298.0 grams)
(P=3.576,C=7.45,F=0.596,E=8.773438917602562)
Total Calories: 2000.0
        Protein: 0.30
        Carbs: 0.17
        Fat: 0.52
```


2. 10% fat

```
Enter name of food data file: nutrients_1000.txt
```

```
        1 - Set maximum protein
        2 - Set maximum carbs
        3 - Set maximum fat
        4 - Exit program

Enter choice (1-4): 3
What percentage of calories from fat is the goal? 10
1: [1.0000] Bread stuffing, bread, dry mix (1 package (6 oz) / 170.0
grams) (P=18.7,C=129.54,F=5.78,E=656.2)
2: [0.3926] Soup, beef and mushroom, low sodium, chunk style (1 cup /
251.0 grams) (P=10.793,C=24.0458,F=5.773,E=68.00000000000001)
3: [1.0000] Teff, uncooked (1 cup / 193.0 grams)
(P=25.669,C=141.1409,F=4.5934,E=708.31)
4: [1.0000] Cookies, oatmeal, prepared from recipe, with raisins (1 oz
/ 28.35 grams) (P=1.8428,C=19.3914,F=4.5927,E=123.3225)
5: [0.0881] Puddings, banana, dry mix, regular, with added oil (1
package (3.12 oz) / 88.0 grams)
(P=0.0,C=77.792,F=4.4,E=29.998642857144432)
6: [1.0000] Corn flour, masa, enriched, white (1 cup / 114.0 grams)
(P=9.6444,C=87.3126,F=4.2066,E=413.82)
7: [0.1077] Onions, young green, tops only (1 stalk / 12.0 grams)
(P=0.1164,C=0.6888,F=0.0564,E=0.3488571428556497)
Total Calories: 2000.0
        Protein: 0.12
        Carbs: 0.79
        Fat: 0.10
```


LAB ACTIVITY | 1.11.1: Greedy Algorithm: Creating A Meal Plan | 0 / 12

Current file: main.py | Load default template...

```
1  import sys, operator, random
2  from nutrition import Food, MealPlan
3
4  # Constants to be used by the greedy algorithm.
5  NUTRIENT_THRESHOLD = 0.001
6  FRACTION_THRESHOLD = 0.05
7  CALORIE_THRESHOLD = 0.1
8  MAX_CALORIES = 2000
9
10
11 def load_nutrient_data(filename):
12     # Open file, read food items one line at a time,
13     # create Food objects and append them to a list.
14     # Return the list once the entire file is processed.
15     return []
```

```
    return []

def sort_food_list(foods, nutrient):
    # Sort the food list based on the percent-by-calories of the
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program | Input (from above) → main.py (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

1.12 Dynamic programming

Dynamic programming overview

Dynamic programming is a problem solving technique that splits a problem into smaller subproblems, computes and stores solutions to subproblems in memory, and then uses the stored solutions to solve the larger problem. Ex: Fibonacci numbers can be computed with an iterative approach that stores the 2 previous terms, instead of making recursive calls that recompute the same term many times over.

PARTICIPATION ACTIVITY 1.12.1: FibonacciNumber algorithm: Recursion vs. dynamic programming.

```
FibonacciNumber(termIndex) {
  if (termIndex == 0)
     return 0
  else if (termIndex == 1)
     return 1
  else
     return FibonacciNumber(termIndex - 1) +
            FibonacciNumber(termIndex - 2)
}
```

FibonacciNumber(4)

FibonacciNumber(0) called 2 times
FibonacciNumber(1) called 3 times
FibonacciNumber(2) called 2 times
FibonacciNumber(3) called 1 time

```
FibonacciNumber(termIndex) {
   if (termIndex == 0)
      return 0

   previous = 0
   current = 1
   i = 1
   while (i < termIndex) {
      next = previous + current
      previous = current
      current = next
      i = i + 1
   }
   return current
}
```

FibonacciNumber(4)

Previously computed 2 terms are stored in memory.
Each term is computed only 1 time.

Animation captions:

1. The recursive call hierarchy of FibonacciNumber(4) shows each call made to FibonacciNumber.
2. Several terms are computed more than once.
3. The iterative implementation uses dynamic programming and stores the previous 2 terms at a time.
4. For each iteration, the next term is computed by adding the previous 2 terms. The previous and current terms are also updated for the next iteration.
5. 3 loop iterations are needed to compute FibonacciNumber(4). Because the previous 2 terms are stored, no term is computed more than once.

PARTICIPATION ACTIVITY 1.12.2: FibonacciNumber implementation.

1) If the recursive version of FibonacciNumber(3) is called, how many times will be FibonacciNumber(2) be called?

- 1
- 2
- 3

2) Which version of FibonacciNumber is faster for large term indices?

- Recursive version
- Iterative version
- Neither

3) Which version of FibonacciNumber is more accurate for large term indices?

- Recursive version
- Iterative version
- Neither

4) The recursive version of FibonacciNumber has a runtime complexity of O($\mathbf{1.62}^{\boldsymbol{N}}$). What is the runtime complexity of the iterative version?

- O($\boldsymbol{N}$)
- O($\boldsymbol{N^2}$)
- O($\mathbf{1.62}^{\boldsymbol{N}}$)

PARTICIPATION ACTIVITY 1.12.3: Dynamic programming.

1) Dynamic programming avoids recomputing previously computed results by storing and reusing such results.

- True
- False

2) Any algorithm that splits a problem into smaller subproblems is using dynamic programming.

- True
- False

Longest common substring

The ***longest common substring*** algorithm takes 2 strings as input and determines the longest substring that exists in both strings. The algorithm uses dynamic programming. An N x M integer matrix keeps track of matching substrings, where N is the length of the first string and M the length of the second. Each row represents a character from the first string, and each column represents a character from the second string.

An entry at i, j in the matrix indicates the length of the longest common substring that ends at character i in the first string and character j in the second. An entry will be 0 only if the 2 characters the entry corresponds to are not the same.

The matrix is built one row at a time, from the top row to the bottom row. Each row's entries are filled from left to right. An entry is set to 0 if the two characters do not match. Otherwise, the entry at i, j is set to 1 plus the entry in i - 1, j - 1.

PARTICIPATION ACTIVITY 1.12.4: Longest common substring algorithm.

```
LongestCommonSubstring(str1, str2) {
   matrix = Allocate str1→length by str2→length matrix

   for (row = 0; row < str1→length; row++) {
      for (col = 0; col < str2→length; col++) {
         // Check if the characters match
         if (str1[row] == str2[col]) {
            // Get the value in the cell that's up and to the
            // left, or 0 if no such cell
            upLeft = 0
            if (row > 0 && col > 0)
               upLeft = matrix[row - 1][col - 1]

            // Set the value for this cell
            matrix[row][col] = 1 + upLeft
```


		z	y	B	o	o	k	s
	L	0	0	0	0	0	0	0
	o	0	0	0	1	1	0	0
	o	0	0	0	1	2	0	0

```
            }
            else
               matrix[row][col] = 0
         }
      }

      substringLength = Maximum value in matrix
      rowIndex = Row index of maximum value in matrix
      startIndex = rowIndex - substringLength + 1
      return str1→substr(startIndex, substringLength)
   }
```

LongestCommonSubstring("Look", "zyBooks") - returns "ook"

k | 0 | 0 | 0 | 0 | 0 | 3 | 0 |

substringLength = 3
rowIndex = 3
startIndex = 1

Animation captions:

1. Comparing "Look" and "zyBooks" requires a 7x4 matrix.
2. In the first row, 0 is entered for each pair of mismatching characters.
3. In the next row, 'o' matches in 2 entries. In both cases the upper-left value is 0, and 1 is entered into the matrix.
4. Two matches for 'o' exist in the next row as well, with the second having a 1 in the upper-left entry.
5. The character 'k' matches once in the last row and an entry of 2 + 1 = 3 is entered.
6. The maximum entry in the matrix is the longest common substring's length. The maximum entry's row index is the substring ending index in the first string.

PARTICIPATION ACTIVITY 1.12.5: Longest common substring matrix.

Consider the matrix below for the two strings "Programming" and "Problem".

	P	r	o	g	r	a	m	m	i	n	g
P	1	0	0	0	0	0	0	0	0	0	0
r	0	2	0	0	?	0	0	0	0	0	0
o	0	0	?	0	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0	0	0	0	0
l	0	0	0	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0	0	0	0
m	0	0	0	0	0	0	1	?	0	0	0

1) What should be the value in the green cell?

- 0
- 1
- 2

2) What should be the value in the yellow cell?

- 1
- 2
- 3

3) What should be the value in the blue cell?

- 0
- 1
- 2

4) What is the longest common substring?

- Pr
- Pro
- mm

Longest common substring algorithm complexity

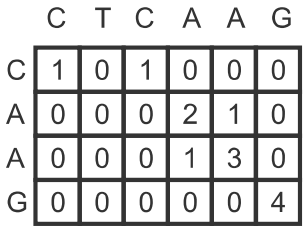
The longest common substring algorithm operates on two strings of length N and M. For each of the N characters in the first string, M matrix entries are computed, making the runtime complexity O($\boldsymbol{N \cdot M}$). Since an N x M integer matrix is built, the space complexity is also O($\boldsymbol{N \cdot M}$).

Common substrings in DNA

A real-world application of the longest common substring algorithm is to find common substrings in DNA strings. Ex: Common substrings between 2 different DNA sequences may represent shared

traits. DNA strings consist of characters C, T, A, and G.

PARTICIPATION ACTIVITY 1.12.6: Finding longest common substrings in DNA.



	C	T	C	A	A	G
C	1	0	1	0	0	0
A	0	0	0	2	1	0
A	0	0	0	1	3	0
G	0	0	0	0	0	4

Longest common substring: CAAG

Uses:

- Finding genetic disorders
- Tracing evolutionary lineages

Dynamic programming advantage:

- Runs with reasonable time complexity

Optimization:

- Store only the following in memory:
  1. Previous row
  2. Location and value of largest matrix entry so far

Animation captions:

1. Finding common substrings in DNA strings can be used for detecting things such as genetic disorders or for tracing evolutionary lineages.
2. DNA strings are very long, often billions of characters. Dynamic programming is crucial to obtaining a reasonable runtime.
3. Optimizations can lower memory usage by keeping only the following in memory: previous row data and largest matrix entry information.

PARTICIPATION ACTIVITY 1.12.7: Common substrings in DNA.

1) Which cannot be a character in a DNA string?

- A
- B
- C

2) If an animal's DNA string is available, a genetic disorder might be found by finding the longest common substring from the DNA of another animal _____ the disorder.

- with
- without

3) When computing row X in the matrix, what information is needed, besides the 2 strings?

- Row X - 1
- Row X + 1
- All rows before X

PARTICIPATION ACTIVITY 1.12.8: Longest common substrings - critical thinking.

1) If the largest entry in the matrix were the only known value, what could be determined?

- The starting index of the longest common substring within either string
- The character contents of the common substring
- The length of the longest common substring

2) Suppose only the row and column indices for the largest entry in the matrix were known, and not the value of the largest or any other matrix entry. What can be determined in O(1)?

- Only the longest common substring's ending index within either string
- The longest common substring's starting and ending indices within either string

Optimized longest common substring algorithm complexity

The longest common substring algorithm can be implemented such that only the previously computed row and the largest matrix entry's location and value are stored in memory. With this optimization, the space complexity is reduced to O(N). The runtime complexity remains O($N \cdot M$).

1.13 Python: Dynamic programming

Dynamic programming for the longest common substring problem

Dynamic programming is a technique where solutions to subproblems are stored in memory and used to quickly find the solution to the full problem.

The longest common substring problem is solved efficiently using dynamic programming. A matrix, a list of lists, stores the length of the longest common substring found in two strings as the strings' characters are examined. Matrix element (i, j) is assigned with zero if the first string's i^{th} character doesn't match the second string's j^{th} character. If the two characters *do* match, then matrix element (i, j) is assigned with the matrix element (i-1, j-1) +1.

Note that string slices in Python use the syntax `str1[ start_index : end_index + 1 ]` to get the substring from start_index up to and including end_index.

Figure 1.13.1: Dynamic programming algorithm for the longest common substring problem.

```
def longest_common_substring(str1, str2):

    # Create the matrix as a list of lists.
    matrix = []
    for i in range(len(str1)):
        # Each row is started as a list of zeros.
        matrix.append([0] * len(str2))

    # Variables to remember the largest value, and the
position it
    # occurred at.
    max_value = 0
    max_value_row = 0
    max_value_col = 0
    for row in range(len(str1)):
        for col in range(len(str2)):

            # Check if the characters match
            if str1[row] == str2[col]:
                # Get the value in the cell that's up and to
the
                # left, or 0 if no such cell
                up_left = 0
                if row > 0 and col > 0:
                    up_left = matrix[row - 1][col - 1]

                # Set the value for this cell
                matrix[row][col] = 1 + up_left
                if matrix[row][col] > max_value:
                    max_value = matrix[row][col]
                    max_value_row = row
                    max_value_col = col
            else:
                matrix[row][col] = 0

    # The longest common substring is the substring
    # in str1 from index max_value_row - max_value + 1,
    # up to and including max_value_col.
    start_index = max_value_row - max_value + 1
    return str1[start_index : max_value_col + 1]
```


PARTICIPATION ACTIVITY 1.13.1: Longest common substring dynamic programming algorithm.

Refer to the code above.

1) Assuming str1 and str2 are both longer than 7 characters, what is up_left's value when row = 7 and col = 0?

- 0
- The value at matrix[7][0]
- The value at matrix[6][0]
- The value depends on what the two strings are.

2) `str1 = "GCCGTATTGT"`
`max_value = 4`
`max_value_row = 6`
Without knowing what str2 is, what is the longest common substring between str1 and str2?

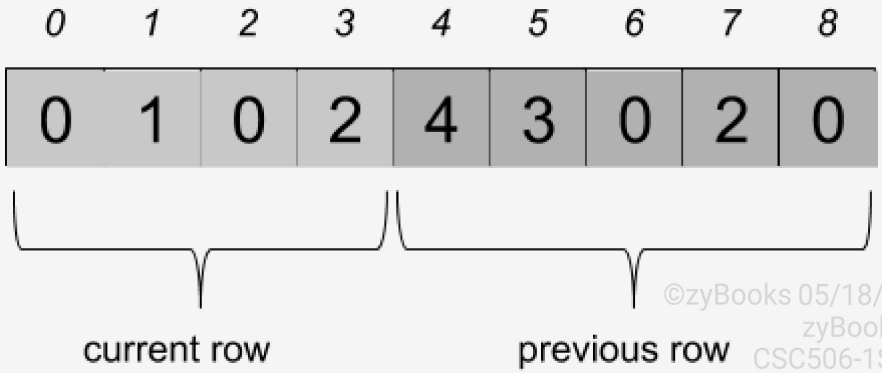
- `GTAT`
- `TTGT`
- Not enough information is given.

3) `str1 = "embrace"`
`str2 = "braille"`
When the algorithm finishes, what is the value of matrix[3][1]?

- 0
- 1
- 2

A space-saving optimization

As the matrix fills in, only the previous row is needed to fill the current row. Thus, the only space required is two rows, plus the variables for the length and row of the longest substring found so far. In fact, once a value is used as an up_left value, the row element can be overwritten as part of the current row. Thus, only a single row is truly needed — the values before the current column are the current row's values, while values after the current column are the previous row's values. Ex: The current column is index 4 with the following row. The indices 0 - 3 (colored green) are from the current row, and indices 4 - 8 (colored red) are from the previous row.



The "2" at index 3 has just been finished, and the value previously at index 3 is stored in variable up_left. The value at index 4 is saved temporarily in the saved_current variable; after the new value at index 4 is put into the row, up_left is assigned with saved_current.

The rest of the algorithm is the same. The optimized function uses $O(N)$ space instead of the $O(N \cdot M)$ space required by the unoptimized function. Both algorithms require $O(N \cdot M)$ time to execute.

Figure 1.13.2: A space-saving optimized version of the longest common substring algorithm.

```
def longest_common_substring_optimized(str1, str2):
    # Create one row of the matrix.
    matrix_row = [0] * len(str2)

    # Variables to remember the largest value, and the row it
    # occurred at.
    max_value = 0
    max_value_row = 0
    for row in range(len(str1)):
        # Variable to hold the upper-left value from the
        # current matrix position.
        up_left = 0
        for col in range(len(str2)):
            # Save the current cell's value; this will be
up_left
            # for the next iteration.
            saved_current = matrix_row[col]

            # Check if the characters match
            if str1[row] == str2[col]:
                matrix_row[col] = 1 + up_left

                # Update the saved maximum value and row,
                # if appropriate.
                if matrix_row[col] > max_value:
                    max_value = matrix_row[col]
                    max_value_row = row
            else:
                matrix_row[col] = 0

            # Update the up_left variable
            up_left = saved_current

    # The longest common substring is the substring
    # in str1 from index max_value_row - max_value + 1,
    # up to and including max_value_row.
    start_index = max_value_row - max_value + 1
    return str1[start_index : max_value_row + 1]
```


PARTICIPATION ACTIVITY 1.13.2: Optimized longest common substring algorithm.

1) The optimized version of the longest common substring algorithm is faster than the original version.

   - True
   - False

2) The optimized version of the longest common substring algorithm requires the the length of str1 is the same as the length of str2.

- True
- False

zyDE 1.13.1: Longest common substring algorithms.

The program below includes both original and optimized versions of the longest common substring algorithm. Try running the program with various inputs to confirm that both algorithms work as expected.

main.py

Load default template...

```
def longest_common_substr
    # Create the matrix o
    matrix = []
    for i in range(len(st
        # Each row is sto
        matrix.append([0]

    # Variables to rememb
    # occurred at.
    max_value = 0
    max_value_row = 0
    for row in range(len(
        for col in range(

            # Check if th
            if str1[row]
                # Get the
```

Look
zyBooks

Run

1.14 Dynamic programming: Get to a location

Specification

Write a program that uses dynamic programming to solve the following problem.

Given a point P (p) on a cartesian plane, take a given amount of steps along the x-axis and y-axis for each iteration and get as close to p as possible. After every 3rd iteration, take a given amount of steps backwards in the x and y directions. Arrive at a point (x, y) whose distance is closest to p (using the distance formula). Start at the origin (0,0).

`Point` class

The `Point` class is provided for you. The class has two data members:

- `x` - The x-coordinate of a point
- `y` - The y-coordinate of a point

The main program

p has been defined and code to read in the x and y coordinates (integers) for point p is also provided.

1) Output p.

2) Read in the number of steps the be taken:

- forwards along the x-axis
- forwards along the y-axis
- backwards along both axes every 3rd iteration

3) Define a dynamic programming algorithm that advances and retreats the required number of steps along the x and y axes and determines the closest point to p. After each iteration, calculate the distance between point p and the current location using the distance function:
`d = sqrt((x_p - x_1)^2 + (y_p - y_1)^2)`
Count the number of iterations. *Hint: Keep track of the previous location.*

4) Output the final arrival point (the point closest to p), the distance between the arrival point and p, and the number of iterations taken.

Ex: For the input

```
4
5
2
3
1
```


where (4,5) is point p, 2 is number of steps to take along the x-axis each iteration, 3 is the number of steps to take along the y-axis each iteration, and 1 is the number of steps to take backwards along both the x and y axes each 3rd iteration, the output is

```
Point P: (4,5)
Arrival point: (4,6)
Distance between P and arrival: 1.000000
Number of iterations: 2
```

Test your program in Develop mode with these and other test input values. Go to Submit mode when you are ready to submit.

*Note: The number of steps to take backwards will never exceed the number of steps taken forwards.

LAB ACTIVITY | 1.14.1: Dynamic programming: Get to a location | 0 / 4

main.py — Load default template...

```
1  import math
2
3  # Point class
4  class Point:
5      def __init__(self):
6          self.x = 0
7          self.y = 0
8
9
10 # Main program
11 # Read in x and y for Point P
12 p = Point()
13 p.x = int(input())
14 p.y = int(input())
15
16 # Read in num of steps to be taken in X and Y directions
17
18 # Read in num of steps to be taken (backwards) every 3 steps
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.py** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


©zyBooks 05/18/24 16:33 1870315
zyBooks User
CSC506-1Spring2024