2-3 Assignment: Code Reflection

Armon Wilson

SNHU

CS-300 Data Structures & Algorithms

**Selection Sort:**

**Purpose:**

A simple sorting algorithm that repeatedly finds the smallest element in the unsorted portion of the list and moves it to the sorted portion.

**Techniques Implemented:**

Uses nested loops to find the index of the minimum element in each iteration, then swaps with the standard swap function.

**Challenges encountered:**

I did not encounter any noteworthy challenges during the implementation of the selectionSort algorithm. All in all, the code is straightforward and easy to understand.

**Pseudocode:**

FUNCTION selectionSort(bids: vector of Bid)

PRINT "Vector Size = ", size of bids

FOR pos = 0 TO (size of bids - 2)

minIndex = pos

FOR j = (pos + 1) TO (size of bids - 1)

IF bids[j].title < bids[minIndex].title THEN

minIndex = j

END IF

END FOR

IF minIndex != pos THEN

swap(bids[pos], bids[minIndex])

END IF

END FOR

END FUNCTION

**Explanation:**

* The function takes a vector of Bid objects (bids) as input.
* It prints the size of the vector.
* It iterates through the vector, starting from the first element (pos = 0).
* For each position (pos), it finds the index (minIndex) of the smallest element in the remaining unsorted part of the vector.
* If the smallest element is not already at the current position (minIndex != pos), it is swapped with the element at the current position.
* The process continues until the entire vector is sorted.

**Partition:**

**Purpose:**

The partition function is a crucial part of the quicksort algorithm. Its primary goal is to rearrange the elements within a given subarray (defined by the begin and end indices) in such a way that all elements with values less than or equal to the pivot element are placed before the pivot and all elements with values greater than the pivot element are placed after the pivot. The function then returns the final index of the pivot element after this rearrangement.

**Techniques implemented:**

* Base Case
  + The condition “(high - low) <= 1” checks if there are zero or one elements remaining in the partition being considered. If so, the pivot is in its final position, and the function returns high.
* Two pointer approach
  + Two pointers, low and high, are used to traverse the subarray from the beginning and end, respectively.
  + “low” moves to the right until it encounters an element greater than or equal to the pivot.
  + “high” moves to the left until it finds an element less than or equal to the pivot.
  + If low is still less than or equal to high, the elements at these positions are swapped. This process continues until the pointers cross.

**Challenges encountered:**

I had a bug within the boundary conditions, where the inner loops (the while loops that find elements to swap) did not have a stopping condition based on low and high crossing each other. This lead to the pointers going out of bounds of the array being partitioned, resulting in crashes when function was invoked.

**Pseudocode:**

FUNCTION partition(bids: list of Bid, begin: integer, end: integer) RETURNS integer

low = begin

high = end

mid = (low + high) / 2

pivot = bids[mid].title

WHILE low <= high

WHILE bids[low].title < pivot

low = low + 1

END WHILE

WHILE pivot < bids[high].title

high = high - 1

END WHILE

IF (high - low) <= 1 THEN

RETURN high

ELSE

swap(bids[low], bids[high])

low = low + 1

high = high - 1

END IF

END WHILE

RETURN high

END FUNCTION

**Explanation:**

* Initialization
  + low and high are set to the begin and end indices, respectively. Mid is calculated as the middle index, and pivot is the title of the element at mid.
* Main Loop:
  + The WHILE loop continues if low is less than or equal to high.
  + The first inner WHILE loop increments low until an element with a title greater than or equal to the pivot is found.
  + The second inner WHILE loop decrements high until an element with a title smaller than or equal to the pivot is found.
  + If high - low <= 1, it means the pivot is in place, so high is returned.
  + Otherwise, elements at low and high are swapped, and low and high are moved closer together.
* The final value of high is returned, indicating the pivot's final position after partitioning.

**Quick Sort:**

**Purpose:**

The primary purpose of the quicksort algorithm is to efficiently sort a collection of elements (in this case, Bid objects by their title) in ascending order.

**Techniques implemented:**

* Recursion:
  + Quicksort is a recursive algorithm, meaning it calls itself to solve smaller subproblems.
* Partitioning:
  + It uses the partition helper function to rearrange elements around a pivot, placing smaller elements to the left and larger elements to the right.
* Pivot Selection:
  + It chooses the middle element as the pivot. (Note: Other pivot selection strategies, like random selection, can be used for better worst-case performance.)
* Optimization:
  + It sorts the smaller of the two sub-arrays first to limit the recursion depth and reduce the risk of stack overflow.

**Challenges encountered:**

I had a bug that caused a stack overflow event. The issue stemmed from the recursive nature of quicksort and how the recursive calls were made. Initially, the code would recursively call itself for both the left and right partitions without considering their sizes. To address this, I modified the quicksort function to prioritize sorting the smaller partition first.

**Pseudocode:**

FUNCTION quickSort(bids: list of Bid, begin: integer, end: integer)

IF begin >= end THEN

RETURN

END IF

mid = partition(bids, begin, end)

IF (mid - begin) < (end - mid) THEN

quickSort(bids, begin, mid)

quickSort(bids, mid + 1, end)

ELSE

quickSort(bids, mid + 1, end)

quickSort(bids, begin, mid)

END IF

END FUNCTION

**Explanation:**

* Base Case
  + If the subarray has 0 or 1 elements, it's already sorted, so the function returns.
* Partition
  + The partition function rearranges the elements and returns the final pivot index.
* Recursive Sorting
  + The function calls itself recursively to sort the sub-arrays before and after the pivot.
* Optimization
  + It determines which sub-array is smaller and sorts it first to minimize recursion depth.