Algorithms & Data Structures I Week 9 Lecture Note

Notebook:	Algorithms & Data Structures I		
Created:	2020-10-21 4:14 PM	**Updated:**	2020-11-04 5:31 PM
Author:	SUKHJIT MANN		

Cornell Notes	**Topic:** Sorting data I	Course: BSc Computer Science Class: CM1035 Algorithms & Data Structures I [Lecture] Date: November 03, 2020

Essential Question:

How do we sort our data and what are the algorithms that make this possible?

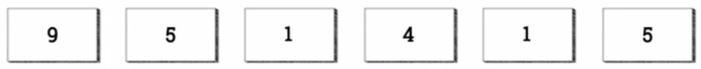
Questions/Cues:

- What are the requirements for sorting data?
- What is the bubble sort algorithm?
- What is the pseudocode for the bubble sort algorithm?
- How can a bubble sort of a vector be implemented on an array?
- How can Bubble sort be applied to a stack?
- What is insertion sort?
    - How are the comparisons made in insertion sort?
- What is the pseudocode for insertion sort?
- What is the array implementation of the insertion sort on a vector?
- How can insertion sort be applied to a stack?
- Describe a vector where insertion sort would require fewer steps in the array implementation than the bubble sort?

Notes

- Sorting data = If we need to sort some data we need to what operations are available to us. So we need to know how our data is collected or what kind of data structure we have: abstract or concrete.
    - But we also need to know what type of data we have in the collection. For example, if the elements are storing numbers, then we can sort numbers from smallest to largest or largest to smallest.
        - If the elements are storing characters such as letters, then we might want to sort the elements according alphabetical order. For other characters coming from different alphabets, we refer to the more general 'lexicographical order'.
    - In computational sorting, we compare one piece of data with another and based on the outcome of this comparison, we can act on our data structure or not
        - Depending on the data structure we would like to sort, how we can make this comparison depends on the operations allowed by that data structure
            - With vectors and dynamic arrays, we can access data in arbitrary locations of our data structure, and compare anything with anything else
            - For a stack or a queue, we can only access data in specific locations, so we are limited in how we can make comparisons

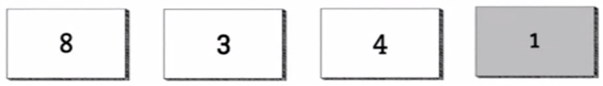
    - Given a vector of integers, let's say we want to sort the elements according to the size of values in them. Let's sort the elements where the smallest value is at the beginning of the vector in element 1 and the largest value is at the final element.



        - Given this information, naturally the first instinct is to start quickly comparing arbitrary elements with others by scanning the values, and then the course of action might be to probably physically move the elements so they are in the right order. Well an important point here to remember is that elements are fixed in place, we don't have operations for picking up elements and moving them around; we can read and write values of elements.
        - Given this constraint, it's possible to simulate the action of picking up two elements and physically swapping them by using a function that we will the swap function (pseudocode below)

```
**function** Swap($vector, i, j$)
    $x \leftarrow vector[j]$
    $vector[j] \leftarrow vector[i]$
    $vector[i] \leftarrow x$
    **return** $vector$
**end function**
```

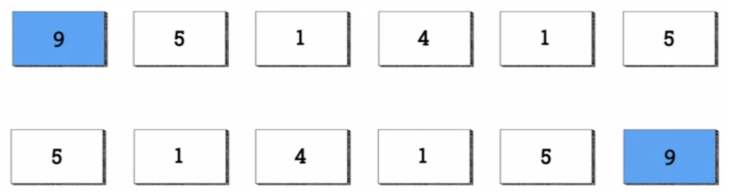
            - The function takes a vector and indices as inputs, and then returns the vector with the values of the two elements with those two indices have been swapped. The basic structure is to create a new variable that stores one of the values at one of the elements, then overwrite that value in the element with the value at the other element. Then finally at the other element, write the value that was stored in the variable
        - Now we have swap function, what remains is to devise a method to compare elements and based on this comparison, swap the values
- Bubble Sort Algorithm (vector example) = In the initial run-through of this algorithm (first-pass), we go along the vector and compare neighboring elements. If the first element has a value that is larger than the value in the next neighboring element, we swap the values using our swap function
    - In order to determine how many such passes are need, we needed to view our algorithm on the most difficult vectors to sort



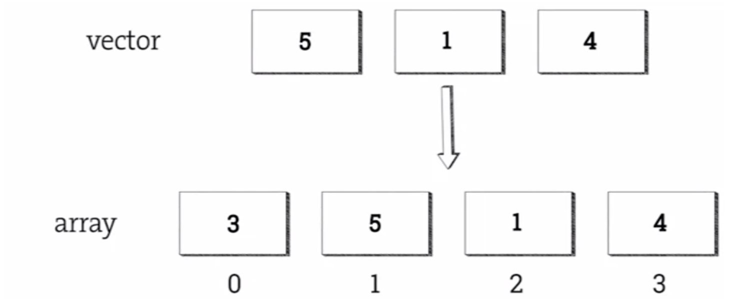
        - Here we see a vector with the smallest integers at the very end, so to fully sort the vector, this value needs to be at the beginning of the vector not the end. At the end of each pass, this value has moved left by one element after a swap is applied. So to get this value at the very beginning where it needs to go, we need as many passes as there are elements in front of this element in the vector. In this case it's three, since the vector is of length four
    - For a larger vector with n elements, then if the smallest value is at the end where the final element is, we would need n - 1 passes to sort the vector

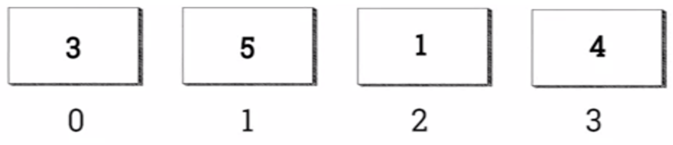
```
function BubbleSort($vector$)
    $n \leftarrow$ LENGTH[$vector$]
    for $1 \leq i \leq n-1$ do
        $count \leftarrow 0$
        for $1 \leq j \leq n-1$ do
            if $vector[j+1] < vector[j]$ then
                Swap($vector, j, j+1$)
                $count \leftarrow count + 1$
            end if
        end for
        if $count = 0$ then
            break
        end if
    end for
    return $vector$
end function
```

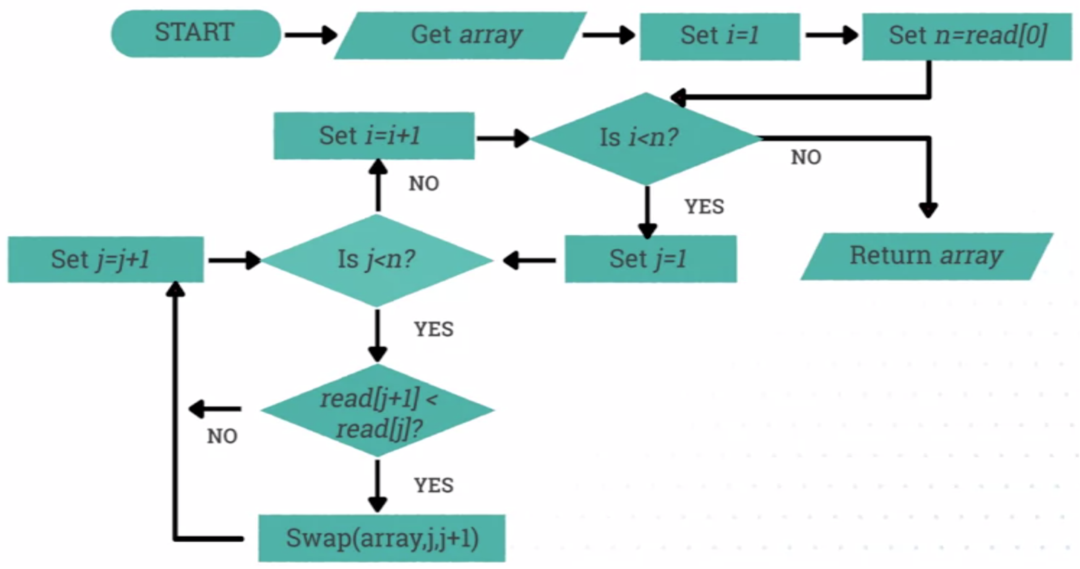
- There are two loops in this algorithm, the first one is quantifying the number of passes being made by the algorithm, and the second loop is the one that compares the neighboring elements of the vector
    - Also there is a counter included that counts the number of swaps that have been implemented. If no swaps have been implemented, this counter remains at zero and the vector is sorted
- As a result, we break out of the first loop and the sorted vector is returned by the function
    - A cool thing to notice about the bubble sort is that at the end of each pass, the largest value in a vector will reach the end in the first pass. This is because we pass along the vector from left to right and the largest value should always go to the right. In this way, we will always be pushing the largest value to the right as we go along



- We see that the bubble sort algorithm will sort the vector because we know the maximum number of passes that are needed to sort a vector by considering the maximum number of swaps that are needed for any particular value
    - The pairwise comparison of elements of the vector make sure that neighboring values are always in the correct order. When the comparison is done enough times, our counter will indicate that no more swaps are needed and the vector is sorted



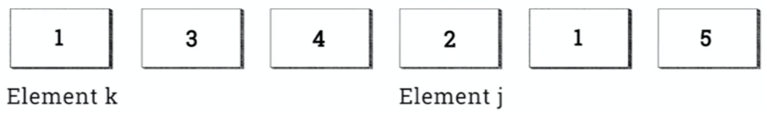




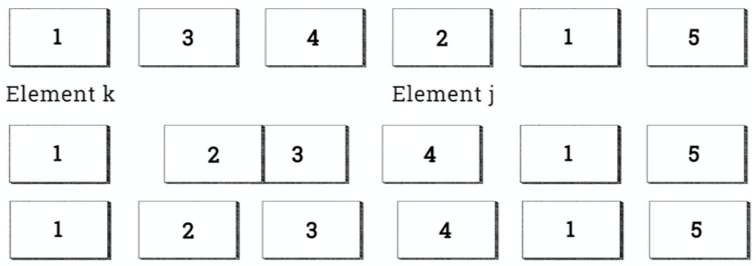
Bubble Sort for Stacks using two stacks

1. Start with two stacks; the stack to be sorted and an empty stack
2. Pop the top element of the stack 1 and push it to the empty stack, effectively making the empty stack, a second stack with a top with the top from the first stack
3. Compare the top elements of the two stacks, mimicking the bubble sort ask, is the top of the second stack smaller than the top of the first stack
    - This is a bit like comparing the neighbors in a vector using the bubble sort algorithm
4. If the top of the second stack is less than the top of the first stack, don't swap the tops and push the top of the first stack to the second stack
5. If the top of the second stack is greater than the top of the first stack, swap the tops of each stack with each other
6. Continue to repeat the previous steps until the largest value is in the top element of the first stack and the only element in that stack. Then push the elements of the second stack into the first stack for the subsequent pass and then continue the process from before
7. Continue to do previous steps until stack is sorted
    - Bubble sort for a stack requires the same number of iterations or passes as the bubble sort for vectored required
        - If we have a stack of length n, we will require n-1 passes of the bubble sort algorithm to fully sort the stack

- Insertion sort = Like bubble sort, the algorithm for insertion sort utilizes lots of pairwise comparisons between elements of data structures. However, the way in which these comparisons are made is different from the bubble sort but there are general structural similarities
    - If we start with a vector of integers as before, and if we go along to each element, instead of comparing it to its next neighbor, we compare it one at a time with all of the previous elements in the vector. This means we start inspecting the elements with the second element of the vector
        - If we find that the element we're inspecting, let's call it element j, has a smaller value than the previous elements but is larger than the value at element k, for being k less than j, then we write the value at element j into element k + 1 and move all other values forward



- In other words, we must write the value to a previous element and shift the values of the other elements along by one, for this we need a shift function (pseudocode below)



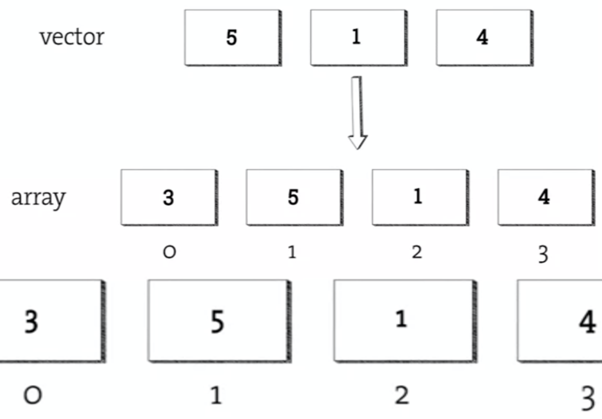
Shift for arrays, dynamic arrays and vectors

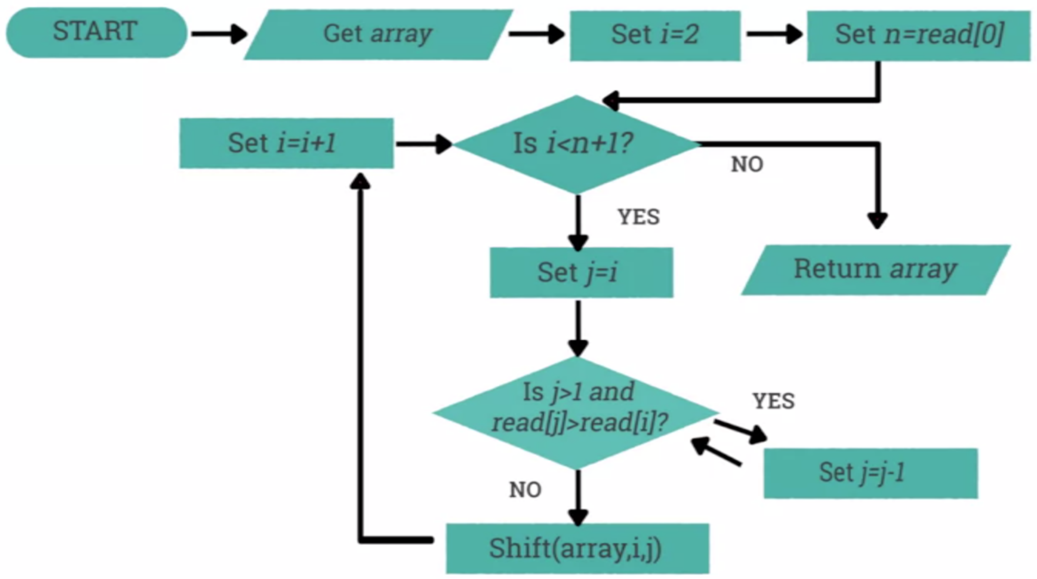
```
function Shift($array, i, j$)
    if $i \leq j$ then
        return $array$
    end if
    $store \leftarrow array[i]$
    for $0 \leq k \leq (i - j - 1)$ do
        $array[i - k] \leftarrow array[i - k - 1]$
    end for
    $array[j] \leftarrow store$
    return $array$
end function
```

- As we can see, the shift function takes an array and two indices as inputs and returns an array with the value at element i being written to element j and all values previously at element j onwards until element i are shifted one place to the right

```
function InsertionSort($vector$)
    for $2 \leq i \leq$ LENGTH($vector$) do
        $j \leftarrow i$
        while $(vector[i] < vector[j - 1]) \wedge (j > 1)$ do
            $j \leftarrow j - 1$
        end while
        Shift($vector, i, j$)
    end for
    return $vector$
end function
```

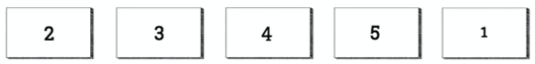
- An observation to be made is that as we go along to each element in the main iteration, all the elements before that element have already been sorted. So we can see how everything will get sorted as we go from element to element





Insertion Sort for Stacks using two stacks

1. Start with two stacks; the example stack being the first stack and an empty second stack
2. Transfer the top of the first stack to the second stack using a pop and a push
3. Compare the two tops of the stacks just like with bubble sort and now we're going to ask, is the top of the second stack larger than the top of the first stack?
4. If the top of the second stack is less than the top of the first stack, we simulate the shift by swapping the tops of the stacks. Then push top of the first stack to the second stack
5. If the top of the second stack is greater than the top of the first stack, don't shift/swap the tops of the stacks and only push the top of the first stack to the second stack
6. Continue to repeat the previous steps until the smallest value is in the top element of the first stack and the only element in that stack. Then push the top of the first stack to the second stack, so that the first stack is empty
7. Now we have an empty first stack and a full second stack, and the second stack is now sorted



- This is a vector of length 5, it is mostly sorted, apart from the element with the smallest value is at the end and we can it at the beginning
- With the bubble sort, we first need to read the length, and then we require four passes in the algorithm, and for each pass, we will make four comparisons and one swap. Each comparison involves two read operations. Each swap involves two reads and two writes. So, in total we will require about 49 read and write operations on the array using the bubble sort
- For the insertion sort, we need to read the length of the vector, then for three iterations make a comparison which involves two reads of the array implementation. So until the last iteration, we

have seven reads. In the last iteration, we need to make eight reads for comparisons. Then to shift everything, we require five reads and five writes at most. So for the insertion sort, we only need about 17 operations in the array implementation in total

- This is the advantage of the insertion sort over the bubble sort. When we have vectors that are very nearly sorted, insertion sort can require far fewer operation in its implementation

Summary

In this week, we learned about requirements for sorting data, what the bubble sort is, what the insertion sort and so much more.