Week 13

Intro to Computer Science - CSI 1430

Hello and welcome to the weekly resources for CSI 1430: Introduction to Computer Science! This week's resource will be covering the course material that will be covered in week 13 of the course.

One of the last major topics we need to talk about is sorting. There are so many different ways we can sort in C++ (or any programming language for that matter) that all have different complexities, efficiencies, runtimes, etc. We will take a look at a couple of these as it will probably come up in your final programs and practicum. **Just as a reminder:** These documents are intended for going over concepts that are discovered in class, as they are a concise and to the point review of the materials. But remember, these documents are not a replacement for lecture and they cannot cover everything, so make sure you get help on any specific concepts you struggle with as well as go over lecture notes, programs, and the textbook.

Reminder: If you have any questions about these study guides, group tutoring sessions, private 30-minute tutoring appointments, the Baylor Tutoring channel or any tutoring services we offer, please visit our website https://baylor.edu/tutoring or call our drop-in center during open business hours. Monday-Thursday 9am - 8pm on class days (254) 710-4135.

Keywords: *Sorting, Selection Sort, Insertion Sort*

TOPIC OF THE WEEK

Sorting

Before we can start to talk about sorting algorithms, I think it's a good idea to define a background as well as a scenario for what is going to be taking place. Let's consider the array of integers as follows:

{ 64, 25, 12, 0, 22, 11 }

Our task is that we want to take this group of unsorted numbers, and create some sort of algorithm to put them into ascending order:

{ 0, 11, 12, 22, 25, 64 }

Selection Sort:

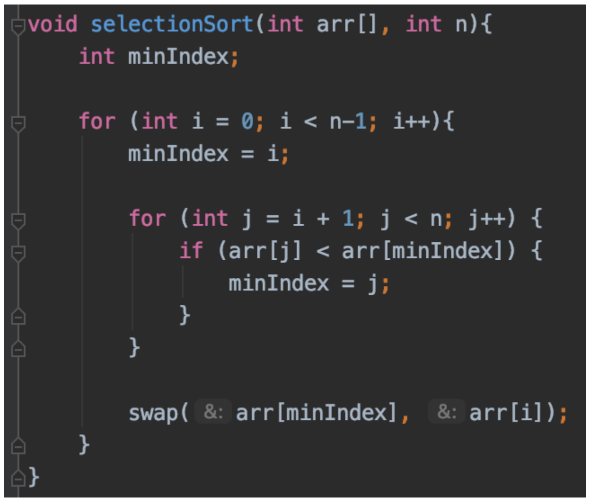
Our first sorting algorithm that we are going to take a look at is the selection sort. Selection sort by definition is a sorting algorithm that sorts an array by repeatedly finding the minimum element of the array within the elements that have not yet been sorted (since we decided on ascending order) and puts it at the beginning, starting from whichever part of the array has not been sorted. During the life of a selection sort algorithm, there are two main parts of the array.

1. The part that has been sorted at the FRONT
2. The remaining elements, yet to be sorted at the BACK

Let's take our unsorted array in our example and apply this logic of the selection sort to it to see what would happen… The blue will indicate the portion that is sorted, and the yellow element is the next found minimum value.

Iteration	Min element found		Shifted to the front	
1:	{ 64, 25, 12, **0**, 22, 11 }	--->	{ 0, 25, 12, 64, 22, 11 }	
2:	{ 0, 25, 12, 64, 22, **11** }	--->	{ 0, 11, 12, 64, 22, 25 }	
3:	{ 0, 11, **12**, 64, 22, 25 }	--->	{ 0, 11, 12, 64, 22, 25 }	*Already in place
4:	{ 0, 11, 12, 64, **22**, 25 }	--->	{ 0, 11, 12, 22, 64, 25 }	
5:	{ 0, 11, 12, 22, 64, **25** }	--->	{ 0, 11, 12, 22, 25, 64 }	
6:	{ 0, 11, 12, 22, 25, **64** }	--->	{ 0, 11, 12, 22, 25, 64 }	DONE!!!

Not only do we need to look at this visually, but also in terms of how we would code something like this in C++. This function definition of the selection sort function takes 2 parameters, being the array of numbers to be sorted, and the size of the array. The initial outside for loop runs through the entire list of numbers is responsible


```cpp
void selectionSort(int arr[], int n){
    int minIndex;

    for (int i = 0; i < n-1; i++){
        minIndex = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        swap( &: arr[minIndex], &: arr[i]);
    }
}
```

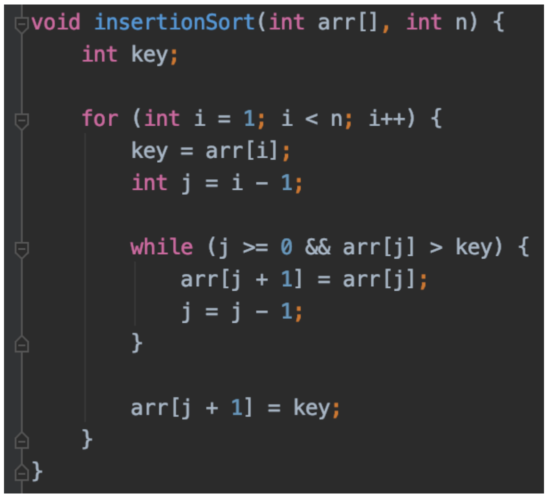
for dividing the array into the 2 groups we talked about in the definition: the sorted front and the unsorted back. It initially starts at 0 since the entire array is unsorted, but every time the loop variable is incremented, then we know that the indexes to the left are sorted, and the right indexes are unsorted. The inner for loop is going to loop through the data within the indexes to the right (the unsorted data), and make a comparison to find which index has the most minimum value...After all the numbers are checked, the 2 indexes are swapped with each other and the cycle repeats until the entire list is sorted.

Insertion Sort:

The next sort we need to talk about is insertion sort. Insertion sort by definition means that we are going to take each element in the array one at a time, from the front to the back, and move it forward until the front subarray (everything to the left of the element that is being moved) is sorted.

Again, let's use our unsorted array in our example and try to walk through the insertion sort for this example... The blue will indicate the front sorted subarray, and the yellow element is the next value in the series to be moved.

Iteration	Current Element		Shifted to correct position	
1:	{ 64, 25, 12, 0, 22, 11 }	--->	{ 25, 64, 12, 0, 22, 11 }	
2:	{ 25, 64, 12, 0, 22, 11 }	--->	{ 12, 25, 64, 0, 22, 11 }	
3:	{ 12, 25, 64, 0, 22, 11 }	--->	{ 0, 12, 25, 64, 22, 11 }	
4:	{ 0, 12, 25, 64, 22, 11 }	--->	{ 0, 12, 22, 25, 64, 11 }	
5:	{ 0, 12, 22, 25, 64, 11 }	--->	{ 0, 11, 12, 22, 25, 64 }	DONE!!!



```
void insertionSort(int arr[], int n) {
    int key;

    for (int i = 1; i < n; i++) {
        key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```

Again, let's take a quick dive into what the code for this looks like. This code example is similar and even a little easier to interpret than selection sort. We take the current element that we are trying to put in the sorted array at the front and while it's not less than a current comparison positions value (that moves forward towards the front) OR if it hasn't reached the front we continue to move forward. Once one of those conditions fails, then we stop...then continue moving forward in the array until all positions in the unsorted array are now sorted.

CHECK YOUR LEARNING

1. **What does the FRONT of a selection sort represent during each iteration?**
2. **What does the BACK of a selection sort represent during each iteration?**
3. **Does the selection sort or insertion sort have a better runtime complexity?**
4. **Although similar, what is the main difference between insertion and selection sort?**

THINGS STUDENTS MAY STRUGGLE WITH

1. One of the things that tends to get brought up when we start learning a whole laundry list worth of new sorting (and searching) algorithms is, WHY? "Isn't it the same if I just learn how to do one and always use that way to sort?!"...Well, not necessarily. Sorting methods tend to have differences in what is called a different runtime complexity. Runtime complexity is essentially how long it will take the computer to perform a specific algorithm. While yes, it is true that if we were sorting a list of 6 numbers every time we programmed a sort, the one we chose would have virtually no effect. HOWEVER, once you start getting into huge amounts of data, then runtime complexity starts to matter a lot more.
2. Just to put the information out there, it might be useful to actually know what the time and auxiliary space of these algorithms are:
   - Selection Sort
     - Runtime Complexity: O(n^2)
     - Auxiliary Space O(1)
   - Insertion Sort
     - Runtime Complexity: O(n^2)
     - Auxiliary Space O(1)

Thanks for checking out these weekly resources! Don't forget to check out our website for group tutoring times, video tutorials and lots of other course resources: https://baylor.edu/tutoring. The answers to the 'Check Your Learning' questions are below.

Answers:

1. The current sorted values.
2. The current unsorted values.
3. Neither! They have the same runtime complexity.
4. Selection sort moves the current minimum value to its correct position, whereas insertion sort moves the next chronological value in the unsorted BACK subarray to it correct sorted position in the FRONT subarray.

Note: All tables were taken from the Baylor CS 1430 zyBook