COP3502 Module 08 Review

Searching

Linear Search – Look through a collection (array, vector, etc.) one element at a time until you find the **target** element.

```
public static int linearSearch (int[] arr, int target)
{
        for (int i = 0; i < arr.length; i++)  // go through array one element at a time
        {
                if (arr[i] = = target)  // if the element (arr[i]) we are on = target
                {
                        return i;     // then return the index of target
                }
        }

        return -1;  // if the whole loop runs and target is never found, return -1
}
```

Worst Case TIME COMPLEXITY:

O(n)

Ex]

linearSearch (| 7 | 2 | 4 | 5 | (0 1 2 3) , 4) would return 2

Binary Search – Repeatedly divide a collection into **smaller and smaller search areas** until the **target** element is found or there is no more left to search (this is called divide and conquer!)

- BINARY SEARCH REQUIRES A SORTED COLLECTION. IF THE COLLECTION IS NOT SORTED, THE ALGORITHM WILL NOT WORK.

The algorithm:

1. Start at the middle index in an array (collection)
2. Ask yourself 3 questions:
   a. **Have I found the target?** If so, you're done! Return the index.
   b. **Is the target bigger than the element I'm looking at?** If so, divide the array (cut it in half) and repeat this whole process with ONLY THE UPPER HALF of the array.
   c. **Is the target smaller than the element I'm looking at?** If so, divide the array (cut it in half) and repeat this whole process with ONLY THE LOWER HALF of the array.

```
public static int binarySearch (int[] arr, int target)
{
        int start = 0, mid, end = arr.length – 1;
        while (start <= end)  // when start becomes > end we have run out of array to search
        {
                mid = (start + end) / 2;  // get the middle element of the array
                if (arr[mid] == target)
                    return mid;
                else if (target > arr[mid])
                    start = mid + 1;
                else if (target < arr[mid])
                    end = mid – 1;
        }

        return -1  //  if the loop finishes running and the target is never found return -1
}
```

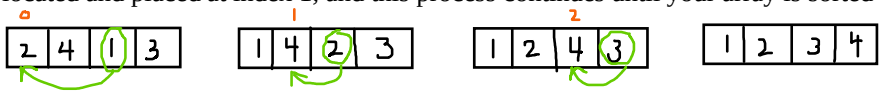
Worst Case TIME COMPLEXITY:

O(log(n))

COP3502 Module 08 Review

Sorting

Selection Sort – Smallest value in array is located and placed at index 0. Then the next smallest value in an array is located and placed at index 1, and this process continues until your array is sorted

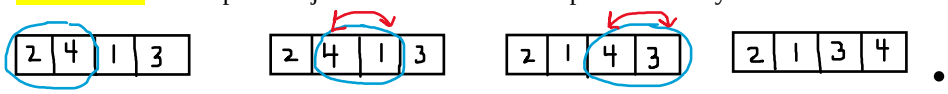


original array | 1st pass | 2nd pass | 3rd pass | There are **size – 1** passes

Worst case time complexity

$O(n^2)$

Bubble Sort – Compares adjacent elements and swaps them if they are out of order



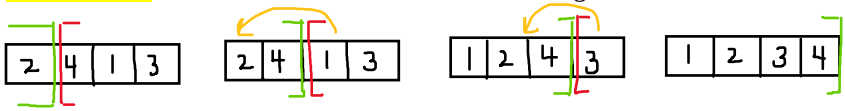
original array | 1st pass | There are **size – 1** passes

Repeat this process

Worst case time complexity

$O(n^2)$

Insertion Sort – Takes each element from unsorted region and inserts it into its correct place in sorted region



1st pass | 2nd pass | 3rd pass | 4th pass | There are **size** passes

Worst case time complexity

$O(n^2)$

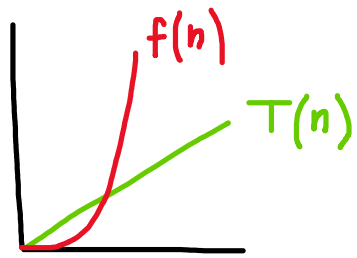
Time Complexity

So, you might be wondering what is all this "O(something)" stuff?

This "O(something) stuff" more formally known as Big-O notation is a way to measure the performance of an algorithm.

Things to know:

- Algorithms (like the searching and sorting algorithms we just discussed) have a **growth rate function T(n)** – a mathematical function that estimates how much time they will take to execute as input size to the algorithm gets very large.
- When we say that an algorithm is O(**f(n)**) it means that **f(n)** is an UPPER bound for the algorithm's growth rate function **T(n)**. In other words, **T(n)** grows SLOWER than **f(n)**



Ex] **n** is O(**n^2**) because **n** grows slower than **n^2**

T(n) <= **f(n)**

COP3502 Module 08 Review

When we say an algorithm is O(n) for example, it means that the time the algorithm takes to execute can be described by a function (some function T(n)) that grows slower than or at the same rate as n (it grows slower than or at the same rate as a linear function).

From best < < < worst here are all of the growth rate functions we will see in this course:

$$1 \quad < \quad \log(n) \quad < \quad n \quad < \quad n*\log(n) \quad < \quad n^2 \quad < \quad n^3 \quad < \quad 2^n \quad < \quad n!$$

best = least time to execute
worst = most time to execute

O(1) – constant time: an algorithm will always execute in the same time regardless of input size

O(n) – describes an algorithm who's performance grows in direct proportion to the size of input

Going through an array using a simple for loop like this: is an example of an algorithm that is O(n)…
The print statement is O(1). It is executed (repeated) n times where n is the size of the inputted array. So, the whole algorithm is O(n).

```
for(int i = 0; i < array.size; i++)
{
        Print(array[i]);
}
```

$O(n^2)$ – describes an algorithm who's performance is directly proportional to the suqare of the size of input
Ex] nested for loops

O(log(n)) – valuable for describing algorithms that divide a problem into smaller sub problems (*hint hint* think binary search!)

Resources to look at:

Lecture slides: https://ufl.instructure.com/courses/436424/pages/module-08-searching-and-sorting-algorithms

Supplementary (hand-written) review page posted under this one ☺