CSE321

INTRODUCTION TO ALGORITHM DESIGN

HOMEWORK 4

Burcu Sultan ORHAN

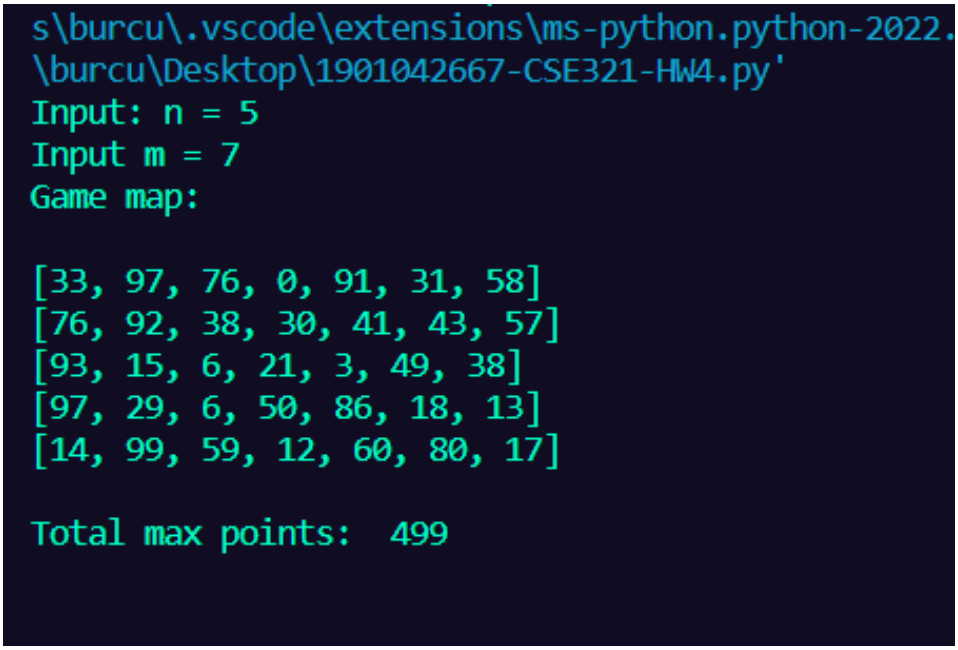
1901042667

Question 1:

In this implementation, I used a recursive method called maxPoints. MaxPoints takes inputs such as: map array, m, n, current location on A axis, current location on B axis, sum of points. This method's base condition is when we reach m and n on A axis and B axis. This means we came to the end of the map and we need to return the sum. Other than that, we need to check whether we came to the end of an axis. For example, if we reached the end of A axis, we can only go right from there. Or if we reach the end of B axis, we can only go down from there. And lastly, if none of these conditions happen, we check if A+1,B is greater than A,B+1 or not. We continue with the greater one, it is added to sum and we call our method again this time with updated locations and sum.

Worst-time complexity: This algorithm doesn't actually depend on conditions. It always does the the same thing, and it's the simplest thing: Comparison and adding. O(m+n)

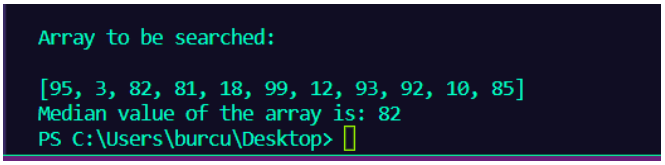
Driver with randomly generated values and output:



Question 2:

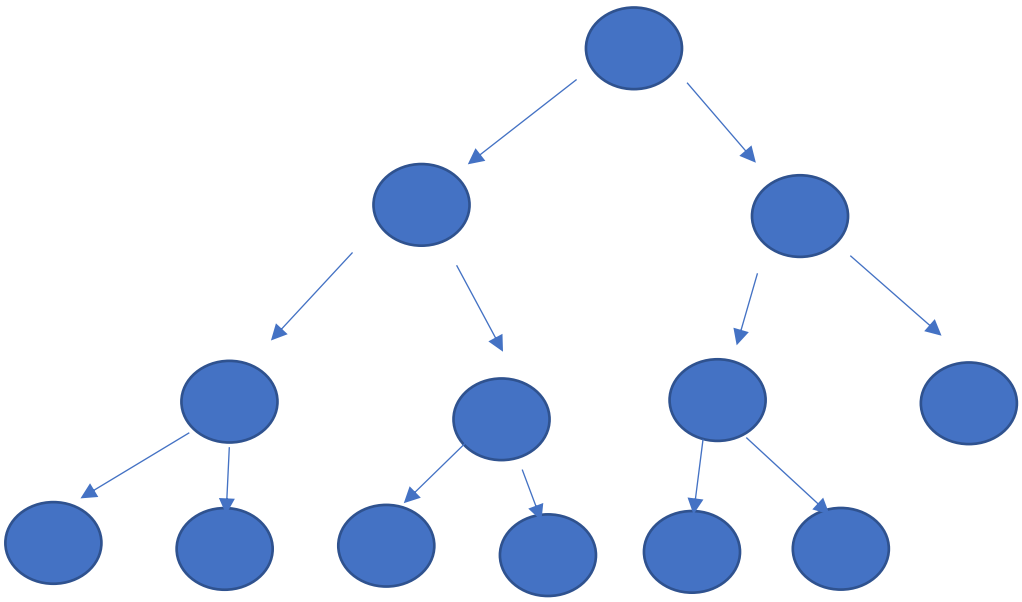
In this implementation, I used an algorithm similar to Quickselect. For that, i needed to implement a helper method for partitioning. Basically my algorithm takes an unsorted array and median value of its number of elements, and firstly sends it to partitioning. In partition, we pick the array's first element as pivot. In this step, i did something not very effective for space usage, which is creating another temporary array for sorting according to the pivot. I did that to avoid swap operations on the original array. And I started to compare pivot to the other elements on the original array. I declared 2 variables as indices for filling the temporary array. i, which is initially 0, counts as index of the elements that will be put into the temporary array as less than the pivot. J, which is initially the length of the array, counts as index of the elements that will be put into the temporary array as greater than the pivot. After filling the temporary array, because I need these changes on the original array, algorithm clears the original array and sets it as temporary array. After these operations, partition method returns the location index of the pivot on the array. After getting the location info of the pivot, main method compares it with median index. If they are equal, that is best case and we found our median value which is pivot. If location is greater than median, that means the median is less than pivot, which must be the array's left side before pivot. For that we call our method again, but this time array is limited as from beginning until pivot. If location is less than median, that means median is greater than pivot, which must be array's right side after pivot. For that we call our method again, but this time array is limited as from pivot until end. Additionally, this time we need to change median value also, because we eliminated the first #pivot values from the array, so we need to find (median-pivot)th smallest element. We work these recursives until we found our median value.

Worst-time complexity: Partition method only makes two for loops sized-n, and no other nested loops or complex operations, so its worst time complexity is O(n). As for main algorithm, worst case happens when the array is reversely sorted in the beginning. Because like this, every pivot will become the current array's greatest element and its left and right sides will be so unbalanced and it will turn into a linear search with every single pivot. Which is O(n^2).



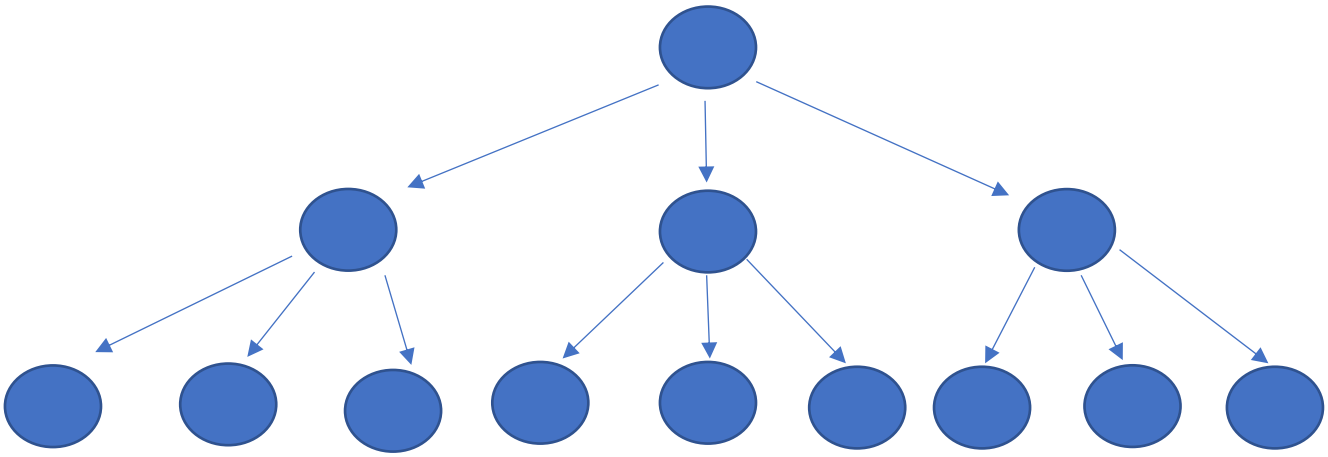
Question 4:

The logic behind Binary tree or Ternary tree is to reduce comparisons as much as we can. I need to visualize to show it better:



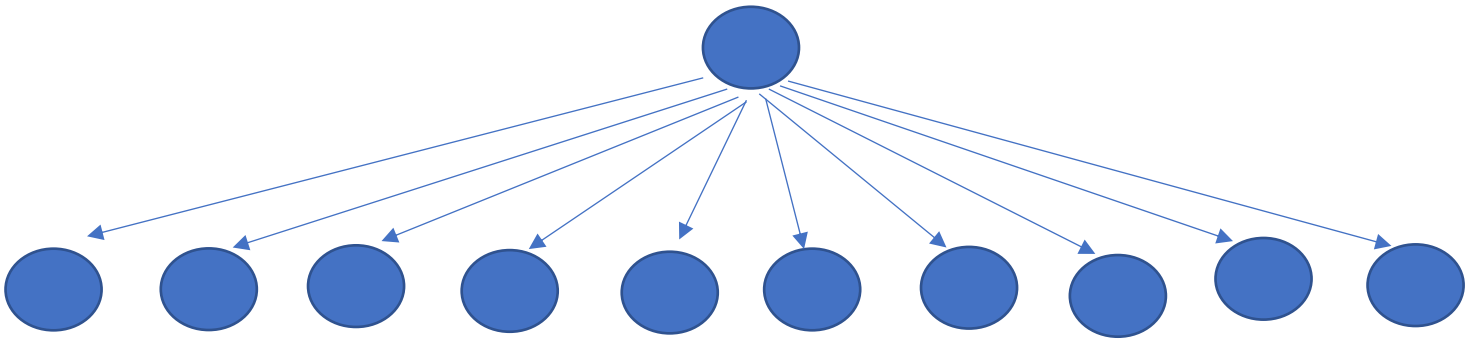
As shown above, at each level, only one comparison occurs. That means, for a list sized 13, we can find the answer by 3 comparisons. This comes from the tree's structure. Such as, level n consists of 2^n elements. Ex: Level 0 has 2^0 = 1 element, level 1 has 2^1 = 2 elements, level 2 has 2^2 = 4 elements. This goes exponential and summation of these element sizes equals to 2^(x) – 1, where x is number of levels, which means number of comparisons (because at each level only 1 comparison occurs). So, number of comparisons is log2(n) + 1, which is element of O(logn).

Lets continue with size 13 list but this time with Ternary search, which is base 3. Only difference is we divide the list into 3 sublists instead of 2. Lets see how it differs with visualization:



As shown above, at each level, 2 comparison occurs. Let me explain this more: in binary tree, we only check if the target equals to the current node or less than or bigger than current node. But in ternary tree, we have 2 more components to divide the list into two, and we compare target with both of them to see if target is less than the small one or greater than the bigger one, if not both, target directs to the middle child. So, at each level 2 comparisons occur. That means, for a list sized 13, 4 comparison occurs. This comes from the tree's structure. Such as, level n consists of 3^n elements. Ex: Level 0 has 3^0 = 1 element, level 1 has 3^1 = 3 elements, level 2 has 3^2 = 9 elements. This goes exponential and summation of these element sizes equals to 3^(2x) – 1, where x is number of levels, which means number of comparisons (because at each level 2 comparisons occur). So, number of comparisons is (log3(n) + 1)/2, which is element of O(logn).

If we divided the n-sized-list into n parts, the benefits of these trees would disappear because it would turn into a simple linear search. For example, lets continue with 13-sized-list:



For this structure, like ternary tree, we need 12 more components to divide the list into 13. And for that 1 level, we need 12 comparisons. 12 comes from n-1, where n is number of elements, so n-1 comparison occurs. Which is element of O(n).

Question 5:

(a) Best case scenerio is when we find the target in first comparison. That is O(1).

(b) Interpolation is an improved version of Binary search. Like binary search, interpolation needs a sorted array in order to work. In binary search, it always goes to median element, but this isn't best practice always. It would be better if we could go to the elements more likely to be near target. That, is what interpolation does. There are many different interpolation methods and one is linear interpolation. The formula for finding a value with linear interpolation is: Key = data-low/high-low. With that key value, we know which way to go to find target. Average time complexity of the interpolation search is log(log(n)).