Digging Into The World Of Sorting.

Python as any other programming language has sorting functions, whether it was already assigned in the package or if we wrote it. Each way is different from the other, some differences can be in how big the function is, how neatly it was written and how fast it can be.

In order to see the variability, we must apply the functions on a Random list to check all of the different integers, from the biggest to smallest, whatever their organization was. And measuring this using the time it function by first setting all the timers to zero, then assigning a specific number of operations or a quantity of numbers if I may say and then measuring the time they too in order to get our wanted result/list.

The aim of this report is to catch the divergent of the sorting functions, taking in account the length of the random lists, the capacity of the function to understand the integers that it was given, and which one is the fastest.

First of all, we start to look at the simplest one “**Bubble Sort**”, which we can see it is the basic easy one, because it sorts with a scanning and swapping.

The first items are being compared to see if they are out of order, if there are n items in the list, then there are n−1 pairs of items that need to be compared on the first pass. It will continually be moved along until the pass is complete. We can see in the graph that I used 400 integers, yet it takes lots of time until it reaches the right sorting way and especially, when we keep adding which in the graph is the worst case of bubble sort O(n^2), while the other lines are constant, they are the lines of getting the highest and lowest values and they show the best case of it O(n).

A screenshot of a cell phone

Description generated with high confidence

Secondly, **Quick sort** which is considered the fastest way of sorting in python, because A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list. And as we can see in the graph, the adding is constant while the get min and max are increasing with time, which means its getting slow. So we conclude that the quicksort function is the fastest The result is O(log(n)), there is no need for additional memory but has it also has a worst case with big numbers, leaving a very uneven partitions and no recursive solution. In big notation, quicksort's worst-case running time is Θ(n^2 ).

A screenshot of a cell phone

Description generated with high confidence

**Binary Search Tree**, on the other hand will start by checking the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, it must be in the upper half. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree.

When we split the list enough times, we end up with a list that has just one item. Either that is the item we are looking for or it is not. The number of comparisons necessary to get to this point is i where n/2^i=1. Solving for i gives us I = log n. The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the binary search is O(log n), which actually is also the best case for it while the worst ones are both O(n^2) and O(n). As we can see in the graph the adding as we said before takes much more time than getting the values of min and max and especially the min, because it describes the best case of the tree while the get max is on an average point.

A screenshot of a cell phone

Description generated with high confidence

Finally, I’m going to talk about the **Heap sort**. It’s a Tree based data structure; this method is able to organize a list of numbers by creating nodes: We need to keep comparing the element with its left and right children because it’s a binary tree. and pushing it downwards until it reaches a point where both its children are smaller than it. In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of O(n log(n)) comparisons and swaps. And in the graph we can see that the add points are increasing slowly but then exponentially after a specific number of operations, and this tells us that at first it was an average case but then turned into a worst case. While the get min get max lines are differentiating but not increasing a lot, which means also it was a good case but not the best for the heap sort function.

A screenshot of a cell phone

Description generated with high confidence

In conclusion, We can notice that I used different numbers of operations according to the velocity of my computer, and also considering how fast did each function operate. After many tries, messed up graphs, and lots of errors; I found out that Heap sort was the fastest and then Quick sort followed by binary search tree and Bubble Sort; Each one of the functions has its own best and worst case, depending on what I got as a result in my graphs and lists. All of the functions sort in different ways, taking what they need from space and time in order to give the final sorted result that we need.