Asgn4: Writeup
Mika Peer Shalem
Mpeersha

Best and Worst Scenarios for Each Sort

Shell Sort

Time complexity in the average cases are O(n*log^2(n)). This means that the performance is proportional to the size of a sorted array. Also, in each iteration, the size of the array to be sorted decreases. This was implemented using the gap function; by the end of the loop, a new gap is being computed. As the number of iterations increases, the gap gets smaller, which leads to the logarithmic time complexity. In the best-case scenario, the inner while loop would not run since the array is already sorted.

The worst-case scenario for shell sort takes about O(n^2) to be completed. This means that the performance of the sorting algorithm is proportional to the square value of the number of elements. Since it has exponential growth, it performs better when there are fewer elements in the array. For example, while 10 elements lead to time complexity of 100, 100 elements would lead to 10000.

The shell sort performs well with small numbers. This is shown in the code when we use shell sort inside the quick sort algorithm. When the number of elements is less than 8, shell sort performs a lot better than quick sort.

Bubble Sort

The best-case scenario is when the given array is sorted, which would lead to time complexity of O(n). In this case, the inner loop would run and would say that the array is already sorted. This would stop the outer loop. Since only one loop fully runs, the best-case scenario is equal to the number of elements.

The worst-case scenario for the bubble sort is O(n^2). This is because the outer loop goes through each element in the array, and the inner loop does the same thing. So, the algorithm would go through all the elements times all the elements. The average case is the same as the worst case. Once again, the sorting algorithm works well for data sets with a small number of elements.

Bubble sort is especially useful when needing to know if an array is sorted. The other sorts we used would need more than O(n) in their best-case scenario. While bubble sort is not efficient when the array is completely out of order or with large data sets, it is good to use to check if smaller arrays are sorted.

Quick Sort

Quick sort best performance is O(n*log n). It is a logarithmic time complexity since in each recursive run, the size of the array gets smaller. There are two recursive calls for each pivot, each handling a small subset of the values in the array. The best case scenario is when the pivot is the middle value since each side of the pivot would have a recursion call with the same number of elements. The average case, which includes the pivot function that was described in the pseudo-code, is also equal to the same number.

On the other hand, if the pivot is found on each edge of the range of elements. In this case, the time complexity is equal to O(n^2). When it happens, one recursion call would be empty, and the other would need to do all the work. For example, if the array was 3,6,1,9,7 and we would have chosen the number 3 to be a pivot. One recursion would handle numbers with an index that is less than the pivot, which is none. The other recursion would go through the rest of the elements. By the end of it, the algorithm would have to go through all the elements twice to reach a sorted state.

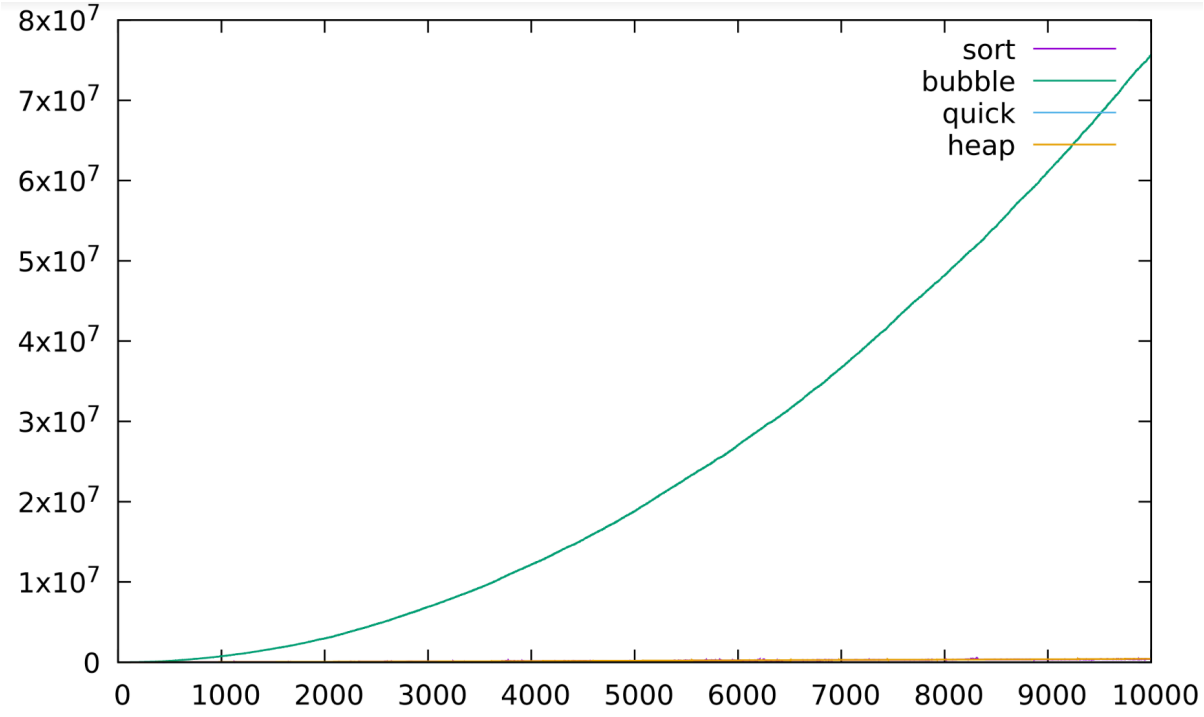
Heap Sort

The time complexity of the heap algorithm is divided into two parts. Building the heap and fixing the heap. The time complexity is equal to O(n*log n). Creating the heap takes O(n), and fixing it takes O(log n). Since we need to add all elements to the heap and then each time we are looking at a specific branch/child of the parent, it is a logarithmic time complexity. It goes through each element, and swaps them if the value of the parent is bigger than the child.

Since heap performs the same for its worst, best, and average cases, it is very reliable. It should be used when there is not a need for a fast sort, but when asked for reliable performance. It also finds the smallest value (in min heap) really quickly. So, it can be used to find the smallest or biggest values.

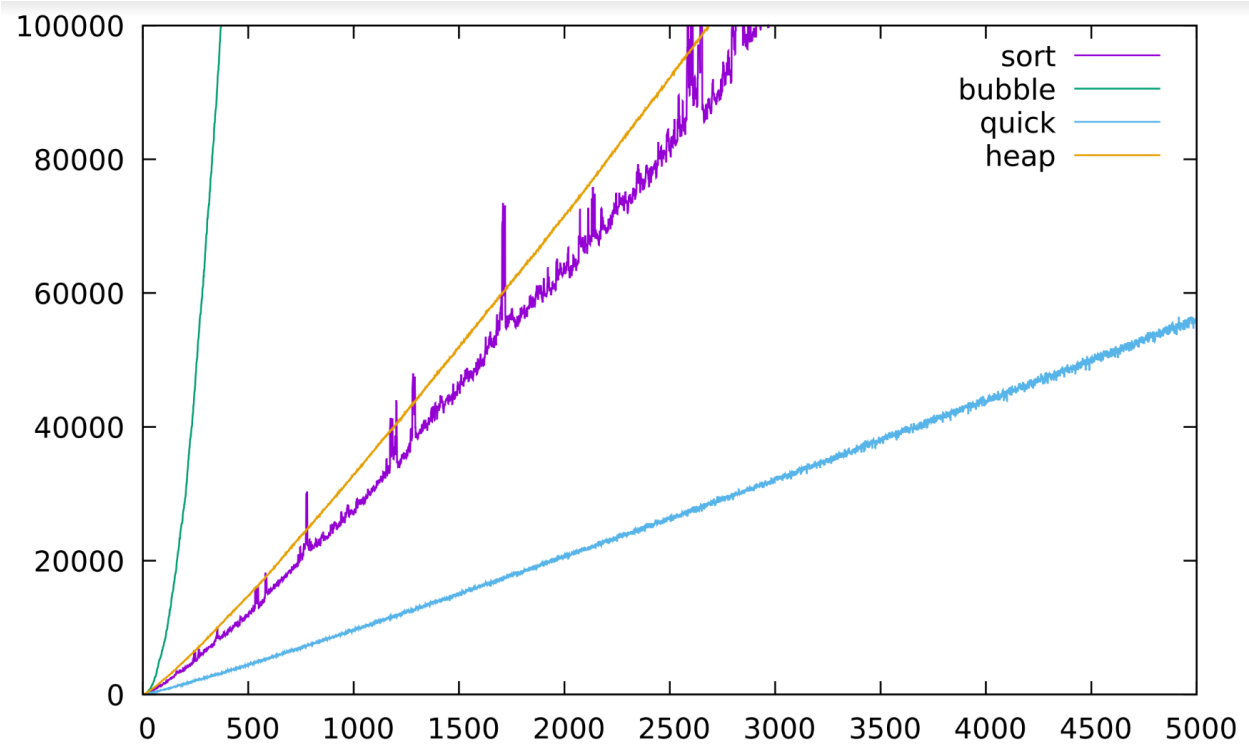
Graphs

At first, I compared the number of elements to the number of moves. In the graphs below, the x-axis represents the number of elements, and the y-axis is the number of moves used by each sorting algorithm. For the first graph, I did not restrict the x- or y-axis. It illustrates how inefficient bubble sort is compared to the other three sort functions. While you can barely see the rest of the functions, bubble sort keeps growing exponentially. It is not helpful in comparing shell, quick, and heap sort. However, it is important to note that bubble sort is not even a close match to the other three algorithms. When the number of elements is less than 500, the sorting algorithms have a similar performance. But, as the number of elements increases, it is possible to see the difference in performances.

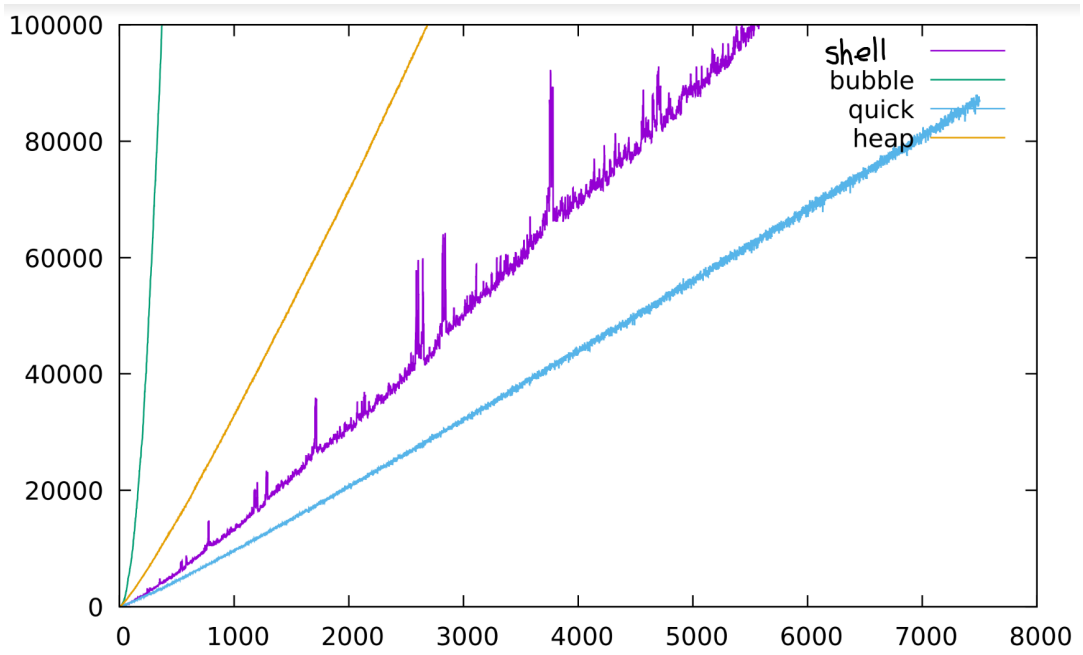


The second graph looks at the same data, but I restricted the x-axis for the first 5,000 elements and the y-axis to 100,000. This way, we can look at the other three sorts. All four sorting algorithms follow the same trend; the length of the array is directly proportional to the number of moves it takes to sort it completely. It is already possible to see that the slope of the bubble sort is a lot steeper than the other three. The shell and heap sort have similar performances, where the heap is usually less efficient, but in some cases, it works better than the shell sort. Overall, quick sort is proving its name by having the least amount of moves.

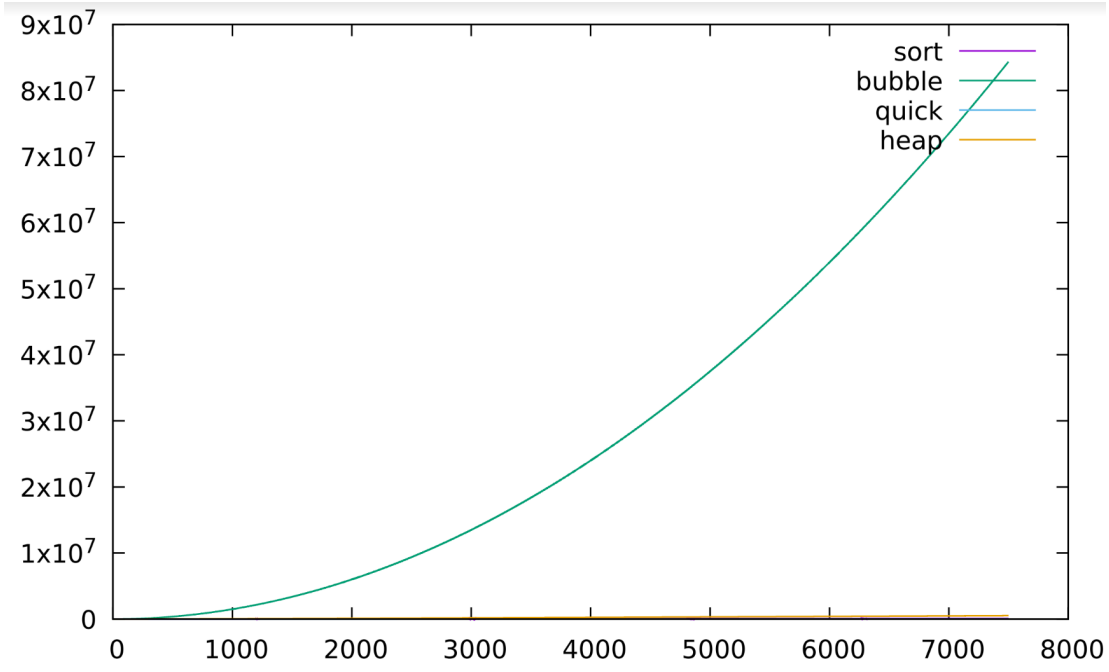
It seems that while heap and bubble act very linearly, shell and quick have more variation. I believe that shell sort acts this way due to the gap function. The same number of elements can have the same gap. For example, 330, 331, and 332 all have a gap of 150. This is because we round down the number we get from the function 5*(number of elements)/11. The overall trend of the function is that as the number of elements increases, the number of moves increases. However, by looking at a smaller range, there would be ups and downs in the gap. Similarly, quick sort is not completely linear due to the recursion the function uses.



I then compared the length of the array to the number of compares. I have restricted the x-axis to be in the middle between the two graphs I described above. The graph demonstrates that the number of comparisons and the number of moves are highly connected to each other. The graphs look almost identical, and it can be due to the difference in the domain. Again, quick sort and shell sort are less linear than the other two. Where shell sort has the most variation in data, and quick sort is almost linear with a smaller variation. However, it shows that heap sort uses a lot more comparisons than shell sort.



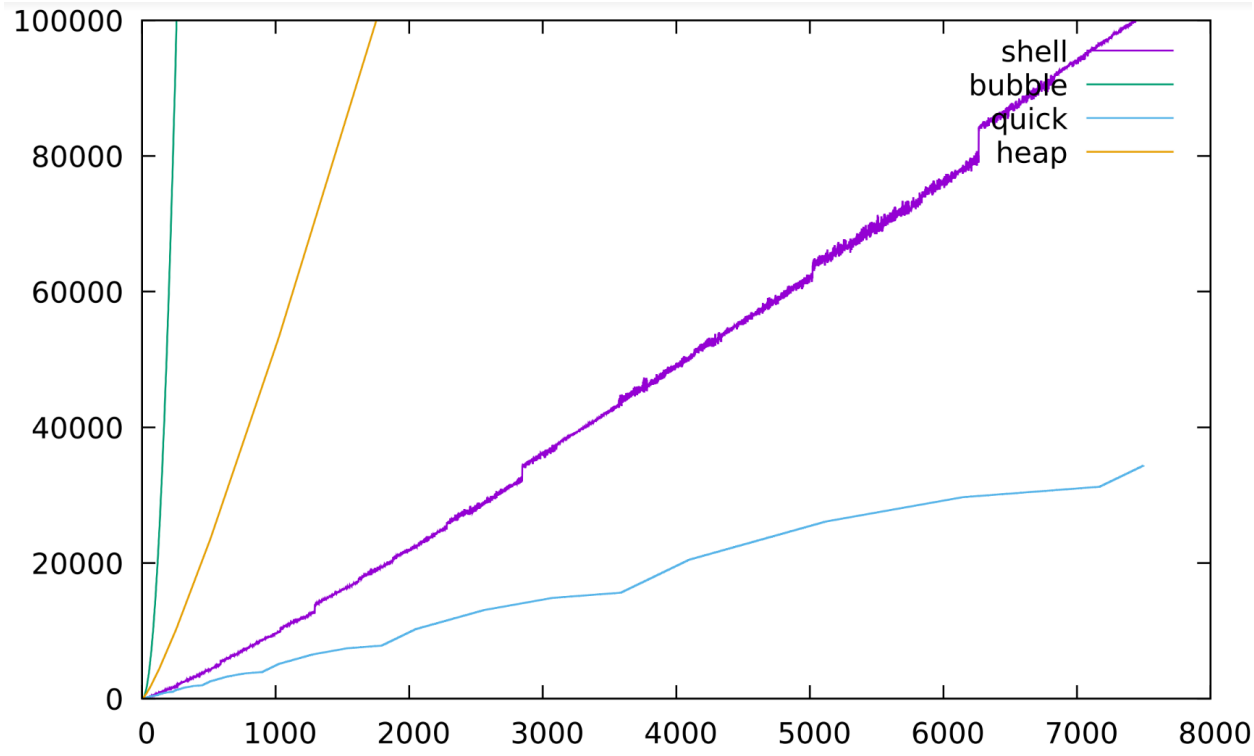
I then created an array in reverse order. I used the random number generator to fill out the array, and then I sorted it so the biggest element would be at index 0 and the smallest number at the last index. This is used to test the efficiency of the sorting algorithms when the array is the opposite of sorted. Similarly to the graphs above, bubble sort still performs the worst when the array is unsorted. It is interesting that bubble sort performs the worst in every case, except when the array is already sorted. It has the best-case scenario and the worst-case scenario at the same time.



I created this graph that also looks at the length of the array vs the number of moves. Here, we can see a difference from when the graph is randomly sorted. Shell sort has the most interesting graph. It grows pretty linearly, but every two thousand elements, the number of moves increases in a non-linear way. It happens for the first time at around 1,400 elements and then happens continuously for the rest of the function. As the number of elements increases, the slope at these points becomes bigger.

Moreover, in this situation, the heap sorting algorithm closely matches the bubble sort rather than the shell sort. This means that heap sort becomes worse as the original array is less sorted since when the original array was random, its performance matches shell sort.

Lastly, quick sort is also affected by this change. Its line graph no longer looks linear, but instead like a connection of many half circles / parabolas. Similarly to shell sort, it has a few points that do not follow the trend. It happens at around 900 elements, 1800 elements, 3,500 elements, and 7,100. It looks like the next point occurs at the last point times two.



What did I learn

The assignment taught me about four different sorting algorithms. Beforehand, I did not realize the various ways to implement sorts. It was interesting to be able to understand the different aspects that go into each sorting algorithm. I believe that now I can recognize when each one should be used. As I described in the first section of the Writeup, each sort has its own benefits and faults. Being able to recognize it makes me a better programmer, since in my future programs I will make sure to take into account the efficiency of my code. I could take the easy route, like bubble sort, or put more effort and reach better results, like quick sort. I was able to use the given stats file to analyze my data. I learned how to use different outputs to compare the sorting algorithms. I had to decide when to compare using the number of moves and when using the number of comparisons.

I also learned how to compute time complexity. I knew that some algorithms are better and faster than others, but I did not know how to calculate this score. The assignment showed me how to analyze given code and find its performance value. If the function has a way that decreases the number of elements in each run, it uses logarithmic time complexity. On the other hand, if there are two loops that go through all the elements, the time complexity is likely to be n^2.

I feel that I understand more about dynamic memory allocation after this practice. The exercise helped me realize the difference between calloc and malloc, as well as when to use each of them. The program works when I use either one of them, but I decided to use calloc since it means that I did not need to initialize each element of the array. I started to use valgrind, and seeing ways to debug my code was very useful. I am planning to continue using these tools in future projects.

The assignment had many small programs, and I learned how to organize my files and divide my time so I could work on all aspects of the project. I was able to effectively test my data to ensure that my program works.

Overall, I am confident with my sorting skills and believe that I would be able to incorporate masking and memory allocation into future projects.

Citations:

1)) Return an array from a function
https://www.tutorialspoint.com/cprogramming/c_return_arrays_from_function.htm

2)) Used to understand how to use structures
https://www.programiz.com/c-programming/c-structures-pointers

3)) Heap Sort Visualization
https://bl.ocks.org/mpmckenna8/5b01b4f6e5cfb8ce022e287c80edaf8d