CME 2204 Assignment-1

Comparison of Heapsort, Shellsort and Introsort

HEAP SORT

This sorting algorithm uses heap logic in the background and sorts by taking the root element of that heap. Therefore, in this algorithm, we also implement the maximum heap algorithm. Parents should always be greater than their children in the maximum heap. So root is the biggest element. Thanks to this feature, the heap is constantly maximized (max-heapify) to ensure that the root element is the largest. When the max heap is complete, the root and last item are swapped (becomes the last item of the array.) and now the last item (max item) is deleted. Repeat this situation while the size of the heap is greater than 1.

- Maximum heap operations n/2. starts from the element. (Rounded to the lower bound.)
- Heap sort is an **in-place algorithm**. That's why memory usage is minimal.
- Heap sort is **not stable**.
- The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. (Very efficient)
- **Time Complexity: Best Case: O(n lgn) , Average Case : O(n lgn), Worst Case : O(n lgn)**

SHELL SORT

Shell sort is a generalized version of the "insertion sort algorithm". It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted. First, the "gap", the distances between the elements to be compared, is calculated. Gap goes from n/2, n/4, n/8... to 1. (The lower bound is taken for n/2 and the following) It starts by comparing the first index with the index n/2 away from it, and it is done for all remaining elements, so the elements are rearranged within the interval (gap). For example, the element in the 4th index is compared with the elements in the 4+(n/2) and 4-(n/2) indexes. Then when it enters the loop again, this time the gap is divided into two and the previous the same operations in the loop are repeated. This process continues until gap(interval) reaches 1 and the array is now completely sorted.

- The performance of shell sorting depends on the type of array used for a given input array.
- Shell sort is an **in-place algorithm**.
- Shellsort is **not stable** because it may change the relative order of elements with equal values.
- **Time Complexity: Best Case: O(nlgn) , Average Case : $O(n(\log n)^2)$, Worst Case : $O(n(\log n)^2)$**

INTRO SORT

Introsort or introspective sort is a hybrid sorting algorithm that provides both fast average performance and (asymptotically) optimal worst-case performance. It begins with quicksort, it switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted and it switches to insertion sort when the number of elements is below some threshold. It also performs insertion sort if the list is small. Introsort is **inplace** and **not stable.**

- The pivot can be dynamically selected as the median or directly as the last element. For very large data, choosing **median slows down** the running time of the Quicksort.
- **Time Complexity: Best Case: O(nlgn), Average Case: O(nlogn), Worst Case: O(nlogn)**

TIME COMPLEXITY

SORTS	BEST CASE	AVERAGE CASE	WORST CASE
HEAP	NLogN	NLogN	NLogN
SHELL	NLogN	$N(\log N)^2$	$N(\log N)^2$
INTRO	NLogN	NLogN	NLogN

	EQUAL INTEGERS			RANDOM INTEGERS			INCREASING INTEGERS			DECREASING INTEGERS		
	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000
heapSort	330600	752700	3262700	523800	1927300	11760700	476300	1703500	10659100	461500	1686900	9484300
shellSort	378800	2432500	9073300	617800	3546500	13320700	391500	2066000	10211500	475900	2557200	8826100
introsort	1179500	2854600	12819500	813200	2430300	16183500	687400	2062700	12840300	1061000	3569300	13095800

- The measurements in the table above are in nanoseconds.

Comparison

For Equal Integers

When we look at this situation, first, heapsort works the fastest in a series of 1000, but shell sort is also running in a close runtime. Intro sorting, on the other hand, works one digit less. As the array grows, the time gap between the heap and other types gets wider. Likewise, the time span in the shell and intro gets wider. As a result, heap sort is always fastest for arrays with equal elements of all sizes, followed by shell and intro sort.

For Random Integers

In this case, the runtime differences in the first array (1000) are not too much. The number of digits are the same and the values are close. When the array becomes large (10k), the runtime of the heapsort is again the lowest, that is, the fastest, but in addition, intro sort runs faster than shell sort. I think because of the intro sort shifts from quick sort to heapsort. When the array gets bigger, it has runtime order as in the first case: heap, shell, intro. In addition, I think one of the most important things is that the values of the runtimes are the same number of digits even if the size of the array increases.

For Increasing Integers

In fact, we can say that this is the best case for all sort algorithms because there is already a sorted array. Therefore, since the best-case runtimes of the algorithms I use are equal, the values are quite close to each other. Of course, there are still very minor differences. For the first array(1k) and for the last array(100k), shell sort is the fastest, while for the 2nd array(10k) heap. Since these times are very close, sometimes their order may change in each study. So, in such cases (incremental) any of these three algorithms can be used.

For Decreasing Integers

We can call this situation a worst case. Because the largest element in the data set is the first element of the array. Looking at the metrics, heap is the fastest for the first array(1k), but the shell is quite close to it. Intro, on the other hand, has a runtime with a higher number of digits as in equal number. In the second array(10k), heap is again the fastest and its difference with the shell becomes clearer. In addition, the intro runtime approaches the shell and heap, but it is still the slowest. In the last array (100k), shell sort is ahead of heap sort and the intro is also getting slower here. Shell can run at the same level as heap, perhaps faster on larger data.

In Addition (Without Best Case)

Looking for heapsort for each array in each state:

- Of all array variants, equal is the fastest case and random is the slowest case. these values are of course close to each other because heap sort is nlogn in all cases.

Looking for shellsort for each array in each state:

- It works in the fastest equal state in 1k arrays. In 10k arrays, it works the fastest in equal case but decreasing state faster than random state. In 100k arrays, it works in the fastest decreasing, the slowest random state.
- The performance of shell sorting depends on the type of array used for a given input array.

Looking for introsort for each array in each state:

- It works in the fastest random state in 1k array. In 10k array, it works the fastest in the random state, and the slowest in the decreasing state. In 100k array, it works fastest in equal then decreasing state, slowest in random state.
- The choice of pivot is very important. The pivot can be dynamically selected as the median or directly as the last element. For **very large data**, choosing **median slows down** the running time of the Quicksort.

For Scenario

Considering that this scenario is included in the random integer part of our measurements, heapsort is the fastest algorithm for all array sizes for the random case in my calculations. The Heap Sort algorithm is very efficient. Effective for sorting a large number of items. While the time required to perform heap sort increases logarithmically, other algorithms can grow exponentially slower as the number of items to sort increases. The heap sort algorithm exhibits consistent performance. As with worst-case performance, best-case performance, average-case performance complexity is the same, O (nlog n). Also, the memory usage usage is minimal. and it's important for the scenario here because it's quite big data.

REFERENCES (SEARCH AND INFORMATIONS)

- HeapSort - GeeksforGeeks
- ShellSort - GeeksforGeeks
- IntroSort or Introspective sort - GeeksforGeeks
- Introsort - Wikipedia
- Shell Sort (With Code in Python, C++, Java and C) (programiz.com)
- Heap Sort (With Code in Python, C++, Java and C) (programiz.com)

CODE REFERENCES

- HeapSort - GeeksforGeeks
- ShellSort - GeeksforGeeks
- IntroSort or Introspective sort - GeeksforGeeks