Project 1: Algorithms and complexity

Comparison between Insert, quick, merge and heap sorting algorithms

Kaleb Meta November 2023

**Introduction**

In this project, we're like detectives investigating different ways to organize a bunch of numbers. We've got four main methods: Insertion, Quick, Radix, and Heap sorting. Our mission is to figure out which one works best as we deal with more and more numbers.

To make it a fair test, we're using a bunch of numbers, and we're starting with a small set of 10 and going up to a whopping 100,000. Then, we're keeping an eye on how long each method takes to get these numbers all sorted out.

1. **INSERTION SORT**

with Insertion sort, you kind of start by thinking the first thing in your list is already sorted. Then, you move to the second thing and check if it's in the right spot compared to the first one. If not, you swap them. And you just keep doing this for all the other things.

Basically, you're comparing each thing with all the stuff that came before it, making sure it finds its proper place in the sorted lineup. Insertion sort is cool for smaller sets of data. Like, if you've got a bunch of things but not too many, Insertion sort gets the job done efficiently. It's like the go-to move for sorting when you're dealing with a not-too-huge bunch of elements.

1. **QUICK SORT**

The Quick sort algorithm operates through a systematic process of partitioning and comparison to efficiently sort an array of elements. The pivotal steps in Quick sort can be outlined as follows:

**Selection of Pivot**:

A pivot element is chosen from the array, typically the first element in our context.

Partitioning:

The array is divided into two halves, with one containing elements smaller than the pivot and the other containing elements larger than the pivot.

Comparison and Adjustment:

Each element in the partitions is compared to ensure conformity with the pivot's positioning. Adjustments are made as needed.

**Central Pivot Placement:**

Once the adjustments are complete, the pivot is positioned in the middle, creating a configuration where smaller elements lie to its left, and larger elements to its right.

Recursive Iterations:

The above steps are then recursively applied to each subsequent partition, gradually reducing the partitions to singular elements and achieving a fully sorted array.

The time complexity of partitioning an array with 'n' elements is denoted as O(n). In the best-case scenario, where the pivot equally divides the array in each recursive call, the time complexity becomes O(nlogn). However, Quick sort can face challenges in the worst-case scenario, leading to a time complexity of O(n^2), particularly if the pivot consistently lands at the first or last position during recursive partitioning.

It is noteworthy that Quick sort is not a stable algorithm, as it alters the order of duplicate values. Nevertheless, it operates in place, requiring no additional storage.

1. **RADIX SORT**

**Step 1: Understand Binary Digits**

Binary numbers consist of bits (0s and 1s). Each bit represents a position in the number.

**Step 2: Identify the Number of Bits**

Determine the number of bits in each binary number. This is crucial for the sorting process.

**Step 3: Initialize Buckets**

Create two buckets for each bit position – one for 0s and one for 1s.

**Step 4: Distribute Numbers into Buckets**

Go through each bit position, starting from the rightmost bit. Place numbers into the buckets based on the value of the bit at that position.

**Step 5: Collect Numbers from Buckets**

After distributing, gather the numbers back, considering the order of the buckets (0s first, then 1s).

**Step 6: Repeat for Each Bit Position**

Repeat steps 4 and 5 for each bit position, moving towards the leftmost bit.

**Step 7: Final Result**

After going through all bit positions, you'll have a sorted list of binary numbers.

Example:

Consider binary numbers 110, 101, 011, 010.

First Pass (Least Significant Bit):

0s Bucket: 110, 010

1s Bucket: 101, 011

Collect Numbers:

110, 010, 101, 011

Second Pass (Next Bit):

0s Bucket: 010, 011

1s Bucket: 110, 101

Collect Numbers:

010, 011, 110, 101

Result:

010, 011, 101, 110 (Sorted binary numbers)

By applying these steps to your binary numbers, you can demonstrate the radix sort process for sorting binary representations.

1. **HEAP SORT**

Heap sort is a sorting algorithm that treats the input array as a special structure called a heap. In a heap, the first element is considered the root node, and for an array of size 'n', each node at position 'k' less than 'n' is viewed as the parent of nodes at positions (2k+1) and (2k+2), as long as they don't exceed the array size.

Here's how the heap sort algorithm works:

**Initial Heap Formation:**

The algorithm starts by iterating over the array, ensuring that every node is smaller than its parent. If not, their values are swapped to maintain the heap property.

**Root Node as Maximum:**

Once the initial heap is formed, the root node holds the largest value in the entire heap.

**Swap with Last Element:**

The root node is then swapped with the last element in the array, and the size of the heap is reduced by one. This step ensures that the last element is sorted and not considered in further heap operations.

**Re-heapify:**

The algorithm then fixes the heap again, starting from the root node and progressing down to the leaf nodes. This process is necessary because swapping the root node may have disrupted the order of the heap.

Repeat Until Sorted:

The above steps of swapping the root node with the last element, reducing the heap size, and re-happifying are repeated until the size of the heap becomes 0. The array is then sorted from the tail to the head.

Heap sort's strength lies in its ability to achieve a sorted array efficiently. The algorithm is characterized by its in-place sorting nature, making it operate with a minimal need for additional memory.

**METHODOLOGY**

The algorithms were implemented in C++. The results were generated for two categories: sorted array (worst-case for quick sort) and randomly shuffled arrays of values 0, 1, . . . , n − 1.

From the sorting algorithms heap and radix sorting came on top. Even though heap performed well in almost every conditions, Radix also had been efficient for all conditions specifically for data with limited number of range. I suggest that radix sorting comes on top as it has very low probability of having a complexity of O(n2).

* For the average case, all algorithms were tested on arrays of sizes from 10, 20, 30, . . . ,100, 200, 300, . . . ,1000, 2000, 3000, . . . ,10000, 20000, 30000, . . . ,100000.
* In both categories, each algorithm was given the same input data. Each array size was tested 100 times and the average time for each size was considered the actual performance.

**RESULTS**

* Graphs of the result are presented as follows
* For the Avg values of the sorting algorithms insertion sort comes out the worst coming to an exponential O(n2) so for easier look to compare the left 3 we have made this graph excluding insertion.
* As we can see Radix outperforms heapsort in AVG cases. FIG:1
* In the second case of sorted arrays all sorting algorithms come very close to each other heap sort coming as most consistent with higher number of ranges and data.FIG:2

Figure 1. comparison of Quick, Radix and heap sorting

FIG:2 Comparison of all algorithms in sorted arrays as a measure of implementation

**CONCLUSION**

* While quick sort and insertion sort has high possibility of having a worst case scenario of complexity O(n2) and radix sort generally needs a lot of space to run because of the bucket system. Heapsort also has stability and speed problems even though it guarantees O(n log n) time complexity in all cases. Considering this cases I can conclude that radix is most efficient, fast and stable for many cases with its Linear time complexity, O(kn), where k is the number of digits and with its high stability.