Submitted By:
YASHASWI TAMTA

VISUALIZATION OF SORTING TECHNIQUES

A Detailed Comparison Analysis

NOVEMBER 1, 2016

YASHASWI TAMTA
University of Washington, Tacoma

Table of Contents

INTRODUCTION

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement.

Project Description

This project consists of visualization of the previously studied soring algorithms which are explained in detail below. In the previous assignments mapped the performance of the sorting algorithms by generating the pertinent graphs. Like Memory management, running time and other factors. In this project we use the plotting and represent it as an animation so that the performance of these sorting algorithms is visually perceived and evaluated. The imperativeness of visualization, is justified by the ease of comparisons between the speed and efficiency of the various algorithms. In the current world speed is the most important factor but in the terms of large data we have to find a balance between speed and the resource usage. The project comprises of the visual performance of each of the five sorting algorithms.

Insertion Sort

Overview

The insertion sort algorithm is the sort unknowingly used by most card players when sorting the cards in their hands. When holding a hand of cards, players will often scan their cards from left to right, looking for the first card that is out of place. For example, if the first three cards of a player's hand are 4, 5, 2, he will often be satisfied that the 4 and the 5 are in order relative to each other, but, upon getting to the 2, desires to place it before the 4 and the 5. In that case, the player typically removes the 2 from the list, shifts the 4 and the 5 one spot to the right, and then places the 2 into the first slot on the left. This is insertion sort. Unlike other simple sorts like selection sort and bubble sort which rely primarily on comparing and swapping, the insertion sort achieves a sorted data set by identifying an element that out of order relative to the elements around it, removing it from the list, shifting elements up one place and then placing the removed element in its correct location. Follow the step by step process of sorting the following small list.

Selection Sort

Overview

The Selection Sort is a very basic sort. It works by finding the smallest element in the array and putting it at the beginning of the list and then repeating that process on the unsorted remainder of the data. Rather than making successive swaps with adjacent elements like bubble sort, selection sort makes only one, swapping the smallest number with the number occupying its correct position.

Consider the following unsorted data: 8 9 3 5 6 4 2 1 7 0. On the first iteration of the sort, the minimum data point is found by searching through all the data; in this case, the minimum value is 0. That value is then put into its correct place at the beginning of the list by exchanging the places of the two values. The 0 is swapped into the 8's position and the 8 is placed where the 0 was, without distinguishing whether that is the correct place for it, which it is not.

Now that the first element is sorted, it never has to be considered again. So, although the current state of the data set is 0 9 3 5 6 4 2 1 7 8, the 0 is no longer considered, and the selection sort repeats itself on the remainder of the unsorted data: 9 3 5 6 4 2 1 7 8.

For the running time we have $O(n^2)$ for the calls to minindex and O(n) for swap. Therefore the algorithm has an average case and best case of $O(n^2)$

Bubble Sort

Overview

Because of bubble sort's simplicity, it is one of the oldest sorts known to man. It based on the property of a sorted list that any two adjacent elements are in sorted order. In a typical iteration of bubble sort each adjacent pair of elements is compared, starting with the first two elements, then the second and the third elements, and all the way to the final two elements. Each time two elements are compared, if they are already in sorted order, nothing is done to them and the next pair of elements is compared. In the case where the two elements are not in sorted order, the two elements are swapped, putting them in order.

Consider a set of data: 5 9 2 8 4 6 3. Bubble sort first compares the first two elements, the 5 and the 6. Because they are already in sorted order, nothing happens and the next pair of numbers, the 6 and the 2 are compared. Because they are not in sorted order, they are swapped and the data becomes: 5 2 9 8 4 6 3. To better understand the "bubbling" nature of the sort, watch how the largest number, 9, "bubbles" to the top in the first iteration of the sort.

(5 9) 2 8 4 6 3 --> compare 5 and 9, no swap 5 (9 2) 8 4 6 3 --> compare 9 and 2, swap 5 2 (9 8) 4 6 3 --> compare 9 and 8, swap 5 2 8 (9 4) 6 3 --> compare 9 and 4, swap 5 2 8 4 (9 6) 3 --> compare 9 and 6, swap 5 2 8 4 6 (9 3) --> compare 9 and 3, swap 5 2 8 4 6 3 9 --> first iteration complete

The bubble sort got its name because of the way the biggest elements "bubble" to the top. Notice that in the example above, the largest element, the 9, got swapped all the way into its correct position at the end of the list. As demonstrated, this happens because in each comparison, the larger element is always pushed towards its place at the end of the list.

For the running time we have $O(n^2)$. But when the list is already sorted it comes as best case and it becomes O(n)

Merge Sort

Overview

Merge Sort is frequently classified as a "divide and conquer" sort because unlike many other sorts that sort data sets in a linear manner, Merge Sort breaks the data into small data sets, sorts those small sets, and then merges the resulting sorted lists together. This sort is usually more efficient than linear sorts because of the fact that it breaks the list in half repeatedly, thus allowing it to operate on individual elements in only log(n) operations, rather than the usual n 2 . Given the data (4 3 1 2) to sort, Merge Sort would first divide the data into two smaller arrays (4 3) and (1 2). It would then process the sub list (4 3) in precisely the same manner, by recursively calling itself on each half of the data, namely (4) and (3). When merge sort processes a list with only one element, it deems the list sorted and sends it to the merging process; therefore, the lists (4) and (3) are each in sorted order. Merge sort then merges them

into the sorted list (3 4). The same process is repeated with sublist (1 2) --it is broken down and rebuilt in to the list (1 2). Merge Sort now has two sorted lists, (4 3) and (1 2) which it merges by comparing the smallest element in each list and putting the smaller one into its place in the final, sorted data set. Tracing how merge sort sorts and merges the subarrays it creates, makes the recursive nature of the algorithm even more apparent. Notice how each half array gets entirely broken-down before the other half does.

Merge sort has aa running time of O(n log n) as the average and worst case.

Quick Sort

Overview

In some way, the quick sort uses a similar idea to the bubble sort in that it compares items and swaps them if they are out of sequence. However, the idea of the quick sort is to divide the list into smaller lists which can then also be sorted using the quick sort algorithm. This is usually done through recursion. Lists of length 0 are ignored and those of length 1 is sorted.

Quick sort, like Merge Sort, is a divide-and-conquer sorting algorithm. The premise of quicksort is to separate the "big" elements from the "small" elements repeatedly. The first step of the algorithm requires choosing a "pivot" value that will be used to divide big and small numbers. Each implementation of quicksort has its own method of choosing the pivot value--some methods are much better than others. The implementation below simply uses the first element of the list as the pivot value. Once the pivot value has been selected, all values smaller than the pivot are placed toward the beginning of the set and all the ones larger than the pivot are placed to the right. This process essentially sets the pivot value in the correct place each time. Each side of the pivot is then quick sorted.

Ideally, the pivot would be selected such that it was smaller than about half the elements and larger than about half the elements. Consider the extreme case where either the smallest or the largest value is chosen as the pivot: when quicksort is called recursively on the values on either side of it, one set of data will be empty while the other would be almost as large as the original data set. To improve the efficiency of the sort, there are clever ways to choose the pivot value such that it is extremely unlikely to end up with an extreme value. One such method is to randomly select three numbers from the set of data and set the middle one as the pivot. Though the comparisons make the sort slightly slower, a "good" pivot value can drastically improve the efficiency of quicksort.

Quick sort like merge sort has an average case of O(n logn) but in worst case it runs at $O(n^2)$. And the choice of pivot affects the performance of this algorithm. It is strongly advised to not choose pivot as the first element.

Description of Data Sets

Synthetic

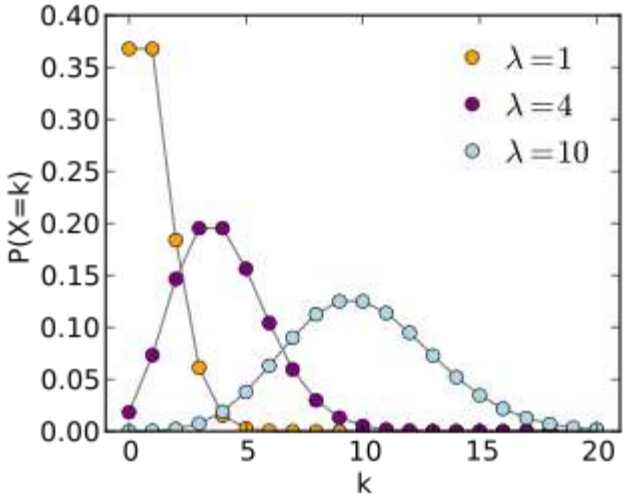
The two Synthetic data sets used are primarily derived from discrete uniform and continuous normal distribution. (Note: a discrete distribution is characterized by a limited number of possible observations as opposed to a continuous distribution which has an infinite number of outcomes.) “A discrete Uniform distribution is a probability distribution whereby a finite number of equally spaced values are equally likely to be observed; every one of n values has equal probability 1/n.” [1] A simple example would be throwing a die.

A continuous Normal Distribution, on the other hand, is a bell-shaped frequency distribution. Here the probabilities found are between two values of the variable. This probability is represented by the area under the curve. The two parameters for the normal distribution are the mean and the standard deviation. The standard deviation gives a measure of the spread of the values around the mean.

Pic taken from:

(http://bestmaths.net/online/index.php/year-levels/year-12/year-12-topic-list/normal-distribution/)

I use a separate function to generate the distribution wherein I pass the id of the desired distribution for example, 1 for Uniform and 2 for Normal. The mean and standard deviation for the normal distribution are calculated randomly therefore, providing a unique distribution in every function call for all the 5 sorting algorithms. I even implemented a Poisson distribution. The Poisson distribution is used to model the number of events occurring within a given time interval where, Lambda is the event rate, also called the rate parameter.



Pic taken from:

(https://en.wikipedia.org/wiki/Poisson_distribution)

Real Data Set

For the real data set I referred the website of UC Irvine Machine learning repository[2]. I took the data set from the above-mentioned website and filtered the data using a self-made python script and chose a specific column with integer values. More is mentioned in the subsection of Real Data Set-1 and Real Data set -2

Synthetic Data Set

For the synthetic data set I had a myriad of options available on the python docs [5]. To help me decide what suits best, I analyzed the output of many distributions available in python before settling for one continuous distribution and one discrete random variable distribution. One other thing I took note of was the degree to which we can control the generation of distribution. Uniform distribution provided the most flexibility and control and therefore was my first choice.

Performance Analysis

The two data sets used comprises of a pseudo random list, and a hard-coded list.

Insertion sort:

This took some time. We can clearly see that the insertion sort first sorts upto a point and then proceeds to the next element. Which is also included in the method and sorted. Therefore, at every iteration the insertion sort is sorted till that element in consideration.

Selection sort:

Selection sort performs fairly. The performance is very well evident that the algorithm is constructed exactly as stated in the definition of selection sort.

Bubble sort:

Just like in the previous assignment Bubble sort takes a lot of time when compared to other algorithms. We can also observe that the higher value propagates till the very end therefore more higher values at the start more iterations and comparisons are required.

Merge sort:

Merge sort performs excellently as it sorts two halves in such a manner that it appears to be simultaneous.

Quick sort:

Quick sort turns out to be fast as well.

Issues and Hurdles:

Initially I worked on animation function of Matplotlib library. This pushed me back a lot of days, trying to draft the working solution was troublesome and tedious. The main alluring fact was that using the FuncAnimation function we could save the animation automatically in an mp4 format. Finally when the code was logically correct some pythonic error came up related to the system, just like last time.

This time though, I could not use someone else's machine to run my tests because it takes a lot of effort to install the required codecs to generate and save animations. This forced me to switch to a more easier method of dynamic plotting. Which turned out wonderfully well.

Another issue came up in the merge sort, but later when it was solved it seemed quite simple. I formulated the solution quite quickly but being new to python it took some time to implement the algorithms of visualizing merge sort which involved list manipulation.

Link:

Visualization of Algorithms - Yashaswi Tamta

References

[1] http://slideplayer.com/slide/1424452/

[2] https://archive.ics.uci.edu/ml/datasets.html

[3] https://archive.ics.uci.edu/ml/datasets/Cylinder+Bands

[4] https://archive.ics.uci.edu/ml/datasets/Iris

[5] https://docs.scipy.org/doc/numpy/reference/routines.random.html