Plist

Pythonic Integer arrays in C

**4th Semester DAA Project**

# Course code: UE17CS251

# By:

Anuj Sanjay Tambwekar

PES1201700056

**Section 4H**

**Department of Computer Science**

**PES University, Bangalore**

# OBJECTIVE

The goal of this project was to implement a structure in C that would be able to emulate the dynamic sizing of properties of lists in python, enhance computational efficiency and implement a C version of Python’s Timsort sorting algorithm.

# IMPLEMENTATION

Python is well known for being a very easy to use language to develop in, allowing for very efficient and fast development. Python’s arrays are contiguous memory allocations that are able to dynamically expand and contract to fit the exact number of elements that the array must contain. Python’s Timsort algorithm is currently one of the best performing sort algorithms that has been developed in the recent years and works by combining principles of insertion sort and merge sort.

C on the other hand is known for being a powerful low level language, but that in turn makes it hard to develop with fluid structures as memory control is one of C’s main features.

The goal of the project is to combine the best of both languages by creating a structure that can :

1. Hold any number of elements (here, integers) in it without needing to have a size declared before hand
2. Dynamically increase or decrease in size
3. Provide additional functionality to help increase algorithmic efficiency

This was implemented by using a Plist structure, that contains the following attributes:

1. An array of integers (list)
2. The maximal element in the list (max)
3. The minimal element in the list (mIn)
4. The number of elements in the (size)
5. A number corresponding to whether or not the array is sorted (sorted)

The functions implemented in the library are:

* Creation of Plist objects
* appending elements
* inserting elements at indexes
* Searching
* deletion at index
* Pop
* Reversing
* sum of all elements in the list
* extending the list
* Reducing to a set
* and the following sort algorithms
  + Bubble Sort
  + Selection Sort
  + Insertion Sort
  + Quicksort
  + Merge Sort
  + Timsort

**Timsort implementation:**

The algorithm implemented is a simplified variant of Python’s Timsort. This implementation does not implement the galloping mode used in the original. The algorithm works by splitting arrays into sub-arrays of length 32, and any runoff is collected in a separate array. This runoff array is sorted by insertion sort, and then each sub array is also sorted by insertion sort. The adjacent subarrays are merged two at a time, with any extra arrays merged with the run-off. This process is repeated till only one array and the runoff array are left, which are finally merged together. It exhibits Θ(nlogn) efficiency for average and worst cases, and a best case efficiency of Θ(n). Due to the additional sort parameter in the structure, the best case is further reduced to Θ(1) for arrays that are already sorted.

# RESULTS

Sorting time comparison for random arrays of various input sizes (time in seconds)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 2K | 10K | 64K | 128K | 256K | 512K |
| Timsort | 0.001 | 0.001 | 0.001 | 0.019 | 0.022 | 0.078 |
| Quick Sort | 0 | 0.001 | 0.001 | 0.019 | 0.021 | 0.062 |
| Merge Sort | 0.001 | 0.002 | 0.01 | 0.021 | 0.047 | 0.094 |
| Insertion Sort | 0.007 | 0.099 | 2.634 | 10.948 | 53.041 | 139.764 |
| Selection Sort | 0.01 | 0.157 | 4.39 | 17.691 | 70.502 | 425.029 |
| Bubble Sort | 0.024 | 0.362 | 13.489 | 55.507 | 137.904 | 1078.08 |

As expected Timsort performs very well as it is an nlog(n) algorithm. Due to the input being entirely random, quicksort does outperform it slightly. However, when comparing arrays that have inherent ordering in them Timsort does much better. The arrays used consisted of 10,000 and 1,00,000 elements comprised of blocks of 50 sorted elements that have been shuffled

Sorting time comparison for semi-sorted arrays of various input sizes (time in seconds)

|  |  |  |
| --- | --- | --- |
|  | 10K | 100K |
| Timsort | 0.001 | 0.006 |
| Quicksort | 0.025 | 0.518 |
| Mergesort | 0.001 | 0.011 |
| Insertion Sort | 0.054 | 5.617 |
| Selection Sort | 0.091 | 9.442 |
| Bubble Sort | 0.164 | 17.127 |