Sorting Algorithms

Contact: "jasvinderahuja@gmail.com"

Types of sorting algorithms

Note: We are assuming an ascending sort

1. Monkey Sort also called Bogo Sort

```
#Pseudocode
While not sorted(array):
    shuffle(array)
return array
```

Time Complexity = $\theta((n-1)*n!)$ *Just don't try it!*

2. Selection Sort

```
#Pseudocode
for an array of length = n select the minimum element put it in position-1
select the 2nd min element for position-2
select the 3rd min element for position-3
.
.
select the nth min (or max) element for position-n
return array
```

Time Complexity = Asymptotic complexity = O(n^2)
Stable: NO

3. Bubble Sort

```
#pseudocode
n=length(array)
repeat n times
    go from index = 0 to index = n
        if array[index-1] > array[index] - swap them
return array
```

Asymptotic Complexity = O(n^2)

4. Insert Sort

```
#pseudocode
n=length(array)
```

```
for each element in array:
    move left to position where element to its left is smaller than itself
return array
```

Versions: Recursive (top-down), Shift (not swap), iterative (bottom-up)

Time Complexity

- Worst case = $O(n^2)$ = average case
- Best case = O(n)
  Stable: Yes

5. Merge Sort

John von Neumann (1945)

Split the array in half recursively, till it is split in one element each. **Apply** sorting and **combine** left and right halves progressively.

```
#pseudocode
function Split(A):
#recursively split
#A=array to sort
    len_a = length(A)
    if len_a <= 1
        return A
    Left = MergeSort[left_half(A)]
    Right = MergeSort[right_half(A)]
    return MERGE(Left, Right)
```

```
function Merge(L,R):
#sort and combine
While i<=length(L) and j<= length(R)

    if L[i] == R[j]:
        new_arr = new_arr.append(L[i])
        new_arr = new_arr.append(R(j])
        i++; j++

    if L[i] < R[j]:
        new_arr = new_arr.append(L[i])
        i++

    if L[i] == R[j]:
        new_arr = new_arr.append(R[j])
        j++
```

```
    if i < length(L)
        new_arr.append(L elements i and beyond)
    if j < length(R)
        new_arr.append(R elements j and beyond)
return merged
```

Time complexity = O(n log n)
Memory: MergeSort is not in place and requires extra memory.
Stable: Yes

Variant: TimSort (named after Tim Peters) combines MergeSort & InsertionSort to give best case O(n) time.

6. QuickSort
   Tony Hoare, 1959

Identify a pivot (best practice = random element), use it to **split** the array into - (smaller_than_pivot, equal_to_pivot*,larger_than_pivot) - recursively till it reaches leaf elements. Then **combine** progressively.

```
function QuickSort()
if length(unique(Arr))>=1
    return
identify random element = X
Partition into - Less_than_X, Equal_to_X* Greater_than_X
QuickSort(Less_than_X)
QuickSort(Greater_than_X)
```

**best practice to handle duplicates*
In-place partitioning methods need no auxillary space and come in two flavours -

- Lomuto's Partitioning
- Hoare's Partitioning

Time Complexity:

- Best-case = O(n log n) = Average Case
- Worst-case = $O(n^2)$

Stability: QuickSort is not stable!

7. Tree Sort
   Calls for Abstract Data Type (ADT) (~binary search tree BST)- Video BST

- Min priority queue
- Max priority queue

> array size = n put elements in a max priority queue (max heap)
> remove n elements from the min priority queue and put from right to left

Time complexity: n * builing-BST = O(n log n) Stability: NO

8. Heap Sort

I like heaps but there are some differences from BST BST vs Heap

Anyways, heaps are what we will be using. Examples-

- Min Heap
- Max Heap

> array size = n, heap height= log n
> put elements in a max priority queue (max heap)
> remove n elements from the min priority queue and put from right to left

Some facts about heaps-

- height of binary tree with n elements = O(log n)
- Complexity for insertion = O(log n)
- Complexity to remove max element = O(log n)
- First element that has a child = n/2

Time complexity: n * builing-heap = O(n log n) Stability: NO

Other applications for heaps–

- Emergency priority queue
- Printer printing jobs
- OS process tracking

9. Radix Sort

> #pseudocode CountSort on least significant digit CountSort on 2^{nd} significant digit CountSort on 3^{rd} significant digit . . CountSort on n^{th} significant digit

Time complexity: O(nd), where d is number of digits ~ O(n log n)
Variant: It maybe useful to change from base 10 to base R

10. Counting Sort

> #pseudocode create bucket/bin for each element count number of each element sort bins visit the buckets in order and populate the sorted array

11. Bucket Sort

> #pseudocode setup empty buckets/bin put object in bucket sort each non-empty bucket visit buckets in order and put all elements into the original array

Algorithms categories (examples from sorting)

- Brute Force (MonkeySort, SelectionSort, BubbleSort)
- Stepwise decrease (InsertionSort)
- Stepwise divide (MergeSort, QuickSort)
- Transform (TreeSort, HeapSort)
- Linear-time Sorting (RadixSort, CountingSort & BucketSort)

Applications for Sorting Algorithms

- Faster value search
- Easy identification of duplicates
- Matching items across arrays
- Identify median or K-th value

Time complexity significance

- At 10^8 operations per second it would take 800 years to sort an array of size 20 by MonkeySort.
- Difference between n log n and n^2 – At 10^8 operations per second 320 million elements (US population) would take ~90 seconds to sort at - n log n - and at - n^2 it would take 32.5 years!!

Fun References

Rounding-off errors in matrix processes. A. M. Turing 1947

Some Facts

- Python uses TimSort a version of MergeSort (stable)
- sort (C++) = QuickSort
- stable_sort (C++) = MergeSort.