MA407: Algorithms and Computation
Exercise 5

November 8, 2024

Jay Razdan

---

1. **Exercise 5.1**

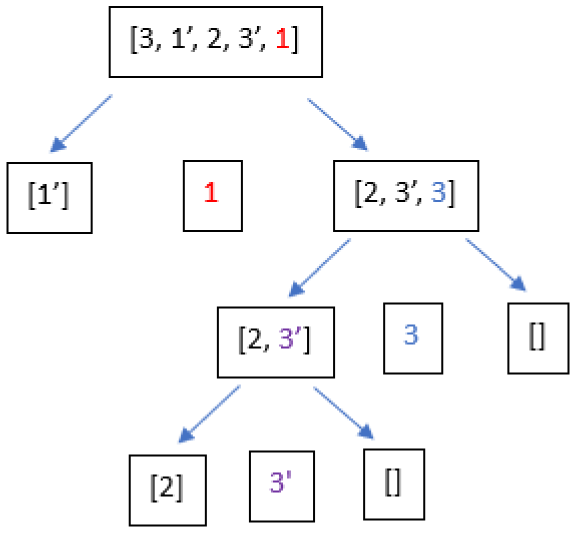
   (a) Give an example showing that the implementation of the **QuickSort** algorithm discussed in the lecture is not stable.

   (b) Suppose you can use additional space. Explain how the Partition procedure can be modified so that **QuickSort** becomes stable.

   (c) Implement the proposed Partition procedure in Python.

Ans:

(a) The **Quicksort** algorithm discussed in the lectures hinged on a "last element" pivot rule and in particular, a partition function to correctly place this given pivot in the sorted list. Let us consider the input list $A := [3, 1', 2, 3', 1]$, and apply the aforementioned **Quicksort** algorithm.

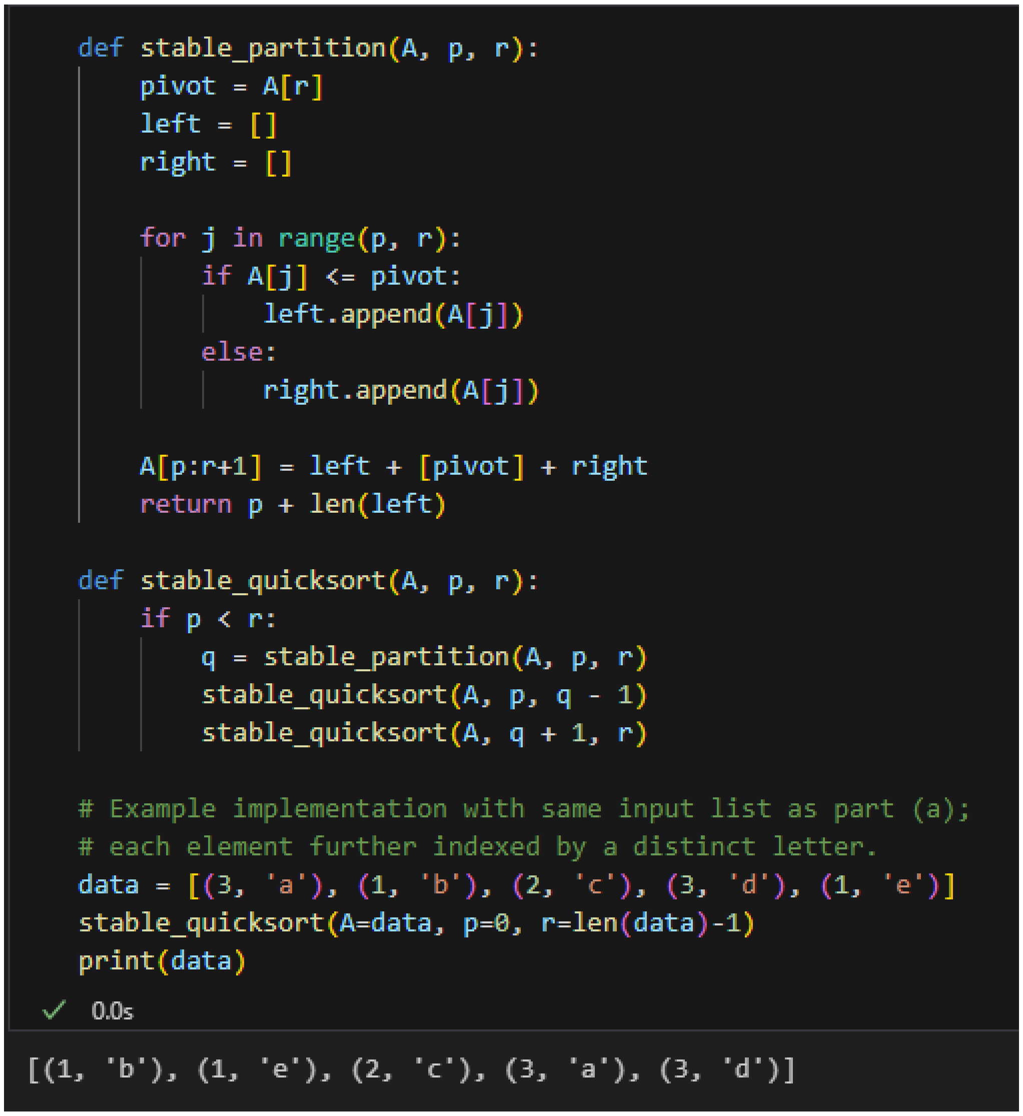


So, the final sorted list outputted by **Quicksort** is: $A = [1', 1, 2, 3', 3]$. However, the sort performed by this **Quicksort** is not stable, because within

the correctly sorted output list, the order of integers with equal value has not maintained its integrity i.e. a stable sort would have resulted in: $A = [1', 1, 2, 3, 3']$, which differs from the **Quicksort** output in terms of the last two elements.
Hence, if we initially hypothesize this **Quicksort** algorithm to be stable, then by means of counter-example, such a **Quicksort** algorithm is not stable.

(b) In order to make the **Quicksort** algorithm stable, we may modify the Partition procedure by involving an additional list, such that we have two lists in total which we append to. By involving an additional list, and making use of extra space, we may place elements less than or equal to, and elements greater than the pivot, in a "left" and "right" list respectively, as we iterate index j throughout the sub-array. In other words, instead of a "swapping" rule between elements to correctly place the pivot, by collecting elements "$\leq$" and "$>$" into respective lists as we iterate in the original order of the input list, we maintain the integrity of the order of elements with equal value when we reconstruct the partitioned (sub-)array by concatenating the "left and "right" lists with the pivot.
Therefore, by using this approach to the Partition procedure, we can make the **Quicksort** algorithm stable.

(c)



```python
def stable_partition(A, p, r):
    pivot = A[r]
    left = []
    right = []

    for j in range(p, r):
        if A[j] <= pivot:
            left.append(A[j])
        else:
            right.append(A[j])

    A[p:r+1] = left + [pivot] + right
    return p + len(left)

def stable_quicksort(A, p, r):
    if p < r:
        q = stable_partition(A, p, r)
        stable_quicksort(A, p, q - 1)
        stable_quicksort(A, q + 1, r)

# Example implementation with same input list as part (a);
# each element further indexed by a distinct letter.
data = [(3, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (1, 'e')]
stable_quicksort(A=data, p=0, r=len(data)-1)
print(data)
```

✓ 0.0s

```
[(1, 'b'), (1, 'e'), (2, 'c'), (3, 'a'), (3, 'd')]
```

2. **Exercise 5.2**

   Consider the variant of **QuickSort** that we discussed in the lecture, in which the Partition procedure picks the last element as pivot.

   (a) Draw the decision tree for inputs A of length $|A| = 3$.

   (b) Over all inputs A of length $|A| = 3$, what is the maximum number of comparisons **QuickSort** needs to carry out to determine the output?

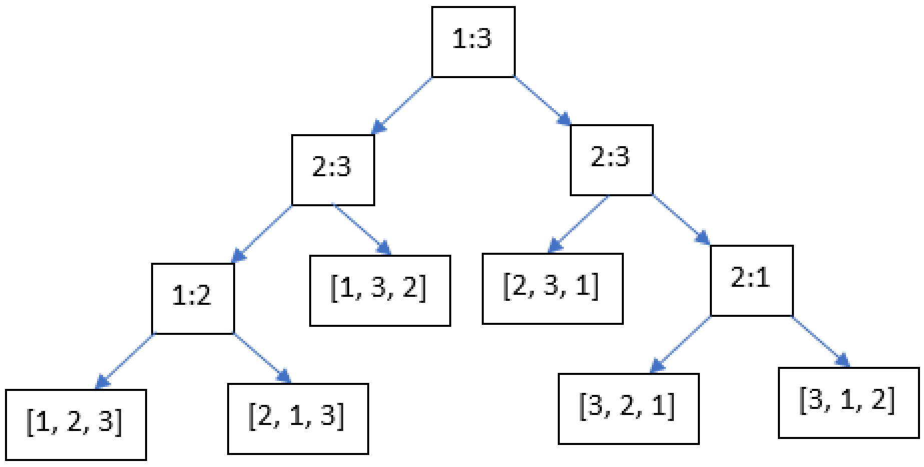
   (c) Give an example input that achieves that maximum number.

   (d) How many reachable leaves are in the decision tree for **QuickSort** on inputs A of length $|A| = 3$?

   (e) For each reachable leaf, give an example input that leads to that leaf.

Ans:

(a) Let us take some initial input array with indices: $[1, 2, 3]$.
The decision tree is as follows:



Note: Left arrows represent the "less than or equal to" decision branch and right arrows represent the "greater than" decision branch.

(b) As we can see from the decision tree, i.e. looking at the number of consecutive arrows, the maximum number of comparisons **Quicksort** needs to carry out to determine the output is: **3**.

(c) An example is: $A := [5, 6, 4]$. In particular, this follows the right-most decision branches of the decision tree depicted in part (a).

(d) Leaves on a decision tree are defined as reachable if there is an input for which the execution of the algorithm corresponds to a path from the root to the leaf, i.e. in regards to the decision tree depicted in part (a), there exists an input for which a set of consecutive arrows leads such an input to a leaf on the tree.
Hence, by observing the drawn decision tree in part (a), we deduce there are **6** reachable leaves.

(e)
1. For leaf $[1, 2, 3]$, input: $A := [4, 5, 6]$
2. For leaf $[2, 1, 3]$, input: $A := [5, 4, 6]$
3. For leaf $[1, 3, 2]$, input: $A := [4, 6, 5]$
4. For leaf $[2, 3, 1]$, input: $A := [6, 4, 5]$
5. For leaf $[3, 2, 1]$, input: $A := [6, 5, 4]$
6. For leaf $[3, 1, 2]$, input: $A := [5, 6, 4]$

3. **Exercise 5.3**

   In the lecture, we discussed **BucketSort**, which sorts n numbers drawn from the *Uniform distribution* on $[0, 1)$, with expected running time $O(n)$.

   (a) Explain why the worst-case running time for **BucketSort** is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \log(n))$?

   (b) We are given n points $p_i = (x_i, y_i)$, for $i = 1, ..., n$, in the unit circle such that $0 < x_i^2 + y_i^2 \leq 1$. Suppose that the points are uniformly distributed over the unit circle: that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Describe and analyse an algorithm with an expected running time of $O(n)$ to sort the n points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$, for $i = 1, ..., n$ from the origin.

   Hint: Design the bucket sizes in **BucketSort** to reflect the uniform distribution of the points in the unit circle.

Ans:

(a) The worst-case running time for **BucketSort** would stem from its worst-case scenario - all the elements of the input array fall into the same bucket. Since **BucketSort**, at least within the scope of the lectures, utilizes the **InsertionSort** to sort each bucket, given that all the elements are only within a single bucket, **InsertionSort** effectively sorts the entire input array as if its elements had never been distributed into buckets in the first place. Finally, since **InsertionSort** has a running time of $\Theta(n^2)$ on an input array of length n, because the worst-case scenario of **BucketSort** follows this eventuality, the worst-case running time for **BucketSort** is: $\Theta(n^2)$.
A simple change to the algorithm to preserve its linear average-case running time and make its worst-case running time $O(n \log(n))$, intuitively, would be to replace **InsertionSort** with a sorting algorithm, on an input array of length n, which instead has a running time of $\Theta(n \log(n))$: **MergeSort** or **HeapSort**.

(b) Since the points are uniformly distributed over the unit circle, we can implement the **BucketSort** algorithm to sort the n points by their distances d_i, for $i = 1, ..., n$ from the origin with an expected running time of $O(n)$. We can achieve this by designing the appropriate bucket sizes such that the expected number of points in each bucket reflects the uniform distribution of points in the unit circle. To do this, we must first notice that the probability of finding a point in any region of the circle is proportional to the area of that region, i.e. a point further out will lie in a region which could contain many

more points than a point lying in a smaller region of the unit circle - we wish to reflect this in the sizes of our buckets. Given that we are sorting by the distances of each point, the bucket sizes will be increasing ranges/intervals of distances for which points in the unit circle will correspond to.
Such a collection of intervals could be that for any k-th bucket, where $k = 0, ..., n-1$, its interval is: $(\frac{k^2}{n^2}, \frac{(k+1)^2}{n^2}]$. This is because for any $k < n$, the ratio/constant of proportionality between their areas is (given they are radii): $\frac{k^2}{n^2}$.
Given we have designed appropriate buckets of equal size (no. of points), implementing the **BucketSort** algorithm will have an expected running time of $O(n)$.