Lecture 10.2 : Quicksort (part 1)

Introduction

- In CA116 you studied *insertion sort* and *selection sort*.
- Here we look at another sorting algorithm called *quicksort*.
- Quicksort is a *recursive* sorting algorithm that employs a *divide-and-conquer* strategy.
- For large problems quicksort significantly outperforms both insertion sort and selection sort.

Partitioning

- Before covering how to implement quicksort we cover one of its core operations: *partitioning*.

```
>>> import random
>>> A = random.sample(range(-99, 100), 20)
>>> len(A)
20
>>> A
[-97, 55, 16, -35, 74, -16, 65, 5, 7, -47, 93, 97, 13, 42, 3, 95, 25, -64, 8, 1
```

- Consider the list of integers `A` above. It consists of 20 random integers in the range [-99,99].
- After partitioning on the element 1, `A` looks like this:

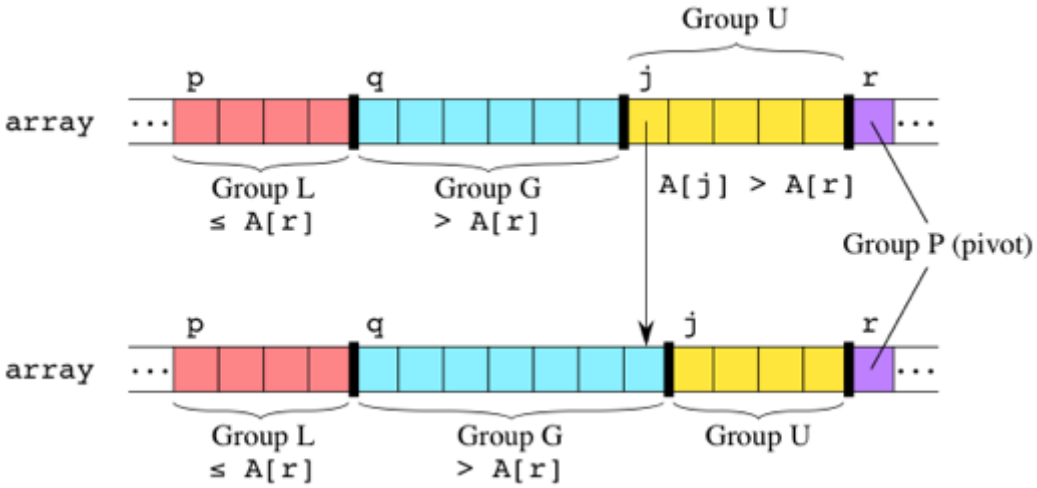
```
>>> A
[-97, -35, -16, -47, -64, 1, 65, 5, 7, 55, 93, 97, 13, 42, 3, 95, 25, 74, 8, 16
```

- The element upon which we partition is called the *pivot*.
- After partitioning on a given pivot:
    - Only elements less than the pivot are found to its left.
    - Only elements greater than the pivot are found to its right.
- Note that the order of the elements to the left and right of the pivot is unimportant. The only thing that matters at this stage is that each element is on the correct side of the pivot.

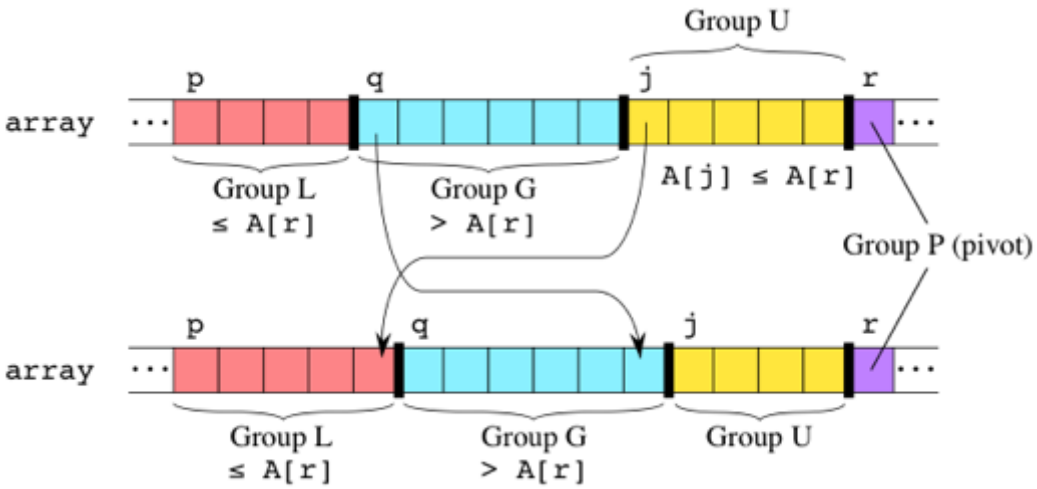
Implementing partitioning

- We will work our way across the list from left to right in order to partition it. Let's call the sequence to be partitioned `A`.
- We need to maintain several indices while doing so: `p`, `q`, `j`, `r`.
- Index `p` is fixed and is the index of the first item in the list to be partitioned (in our case `p` is 0).
- Index `r` is the index of the pivot in the list to be partitioned (in our case `r` is 19).
- Elements in the range `A[p...q-1]` are known to be less than or equal to the pivot.

- Elements in the range `A[q...j-1]` are known to be greater than the pivot.
- Elements in the range `A[j...r-1]` are yet to be processed.
- Initially `q == j == p` since we have not processed anything yet.
- The only indices that are changing are `q` and `j`.
- At each step we compare the element at `A[j]` (i.e. the first element in what remains to be processed) with the pivot `A[r]`.
- If it is greater than the pivot then it is in the correct location so we simply increment `j` (remember `A[q...j-1]` are known to be greater than the pivot) and move on to the next element.



- If `A[j]` is less than or equal to the pivot it needs to move. Where can we put it? Well we can swap it with the element at `A[q]` since the latter is the *first* element known to be greater than the pivot. After the swap we increment `q` (extending the range of items known to be less than or equal to the pivot to take into account the new element). We also increment `j` to move on to the next element to be processed.



- This may sound complicated but implementing it is straightforward:

```
def partition(A, p, r):

    # q and j start at p
    q = j = p

    # up to but not including pivot
    while j < r:

        # move values less than or equal to pivot and update q
```

```
        if A[j] <= A[r]:
            A[q], A[j] = A[j], A[q]
            q += 1

        j += 1

    # swap pivot with A[q]
    A[q], A[r] = A[r], A[q]

    # return pivot index
    return q
```

- Here is the function in action:

```
>>> from partition import partition
>>> A # before partitioning
[-97, 55, 16, -35, 74, -16, 65, 5, 7, -47, 93, 97, 13, 42, 3, 95, 25, -64, 8, 1
>>> partition(A, 0, len(A)-1) # partition
>>> A # after partitioning
[-97, -35, -16, -47, -64, 1, 65, 5, 7, 55, 93, 97, 13, 42, 3, 95, 25, 74, 8, 16
```