Data Structures and Algorithms

Quick Sorting Algorithm

Amilcar Aponte

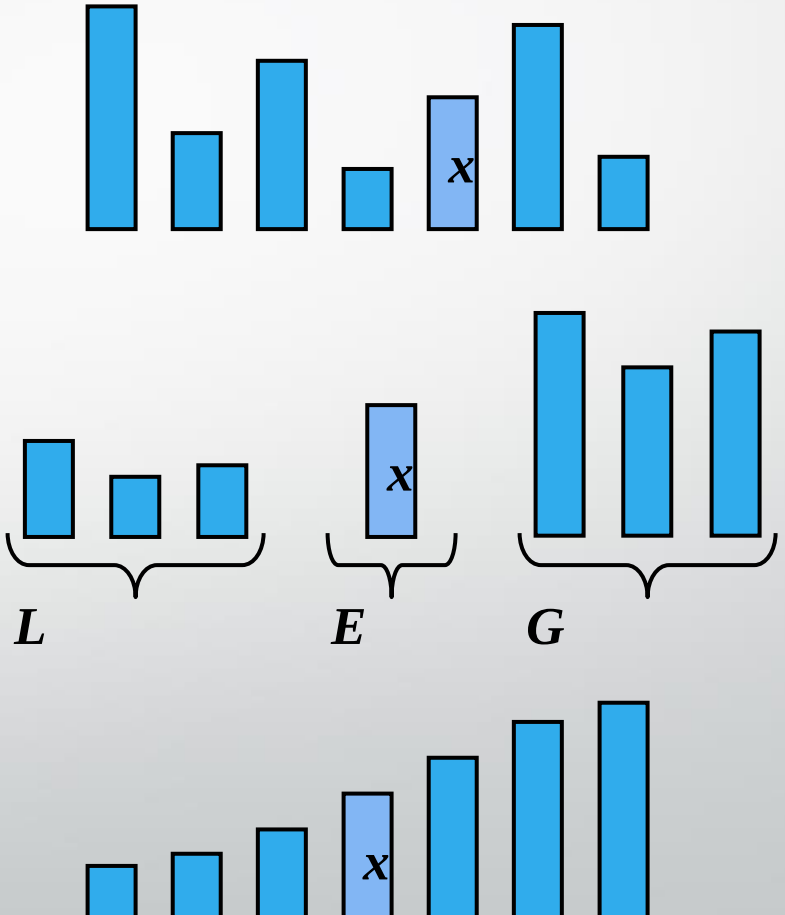
amilcar@cct.ie

Quick Sort

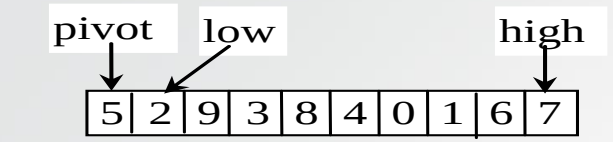
- The algorithm selects an element, called the *pivot*, in the array.
- Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot.
- Recursively apply the quick sort algorithm to the first part and then the second part.

Quick-Sort

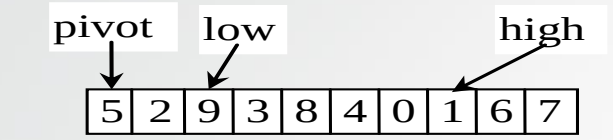
- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



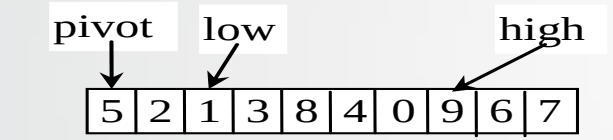
Partition



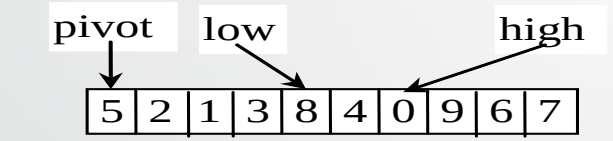
(a) Initialize pivot, low, and high



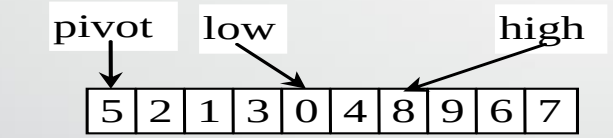
(b) Search forward and backward



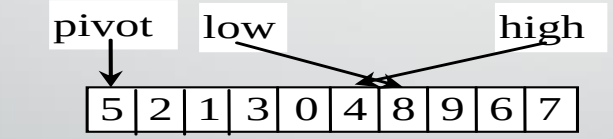
(c) 9 is swapped with 1



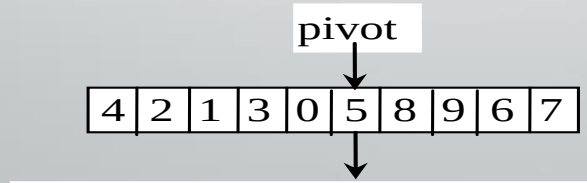
(d) Continue search



(e) 8 is swapped with 0



(f) when high < low, search is over



(g) pivot is in the right place

The index of the pivot is returned

Quick Sort – Pseudo-code

```
quickSort ( array, start, end) {
   if start < end
        pivot_index = partition (array, start, end);  // select pivot and re-arrange elements in two
                                                      // partitions such as
                                                      // all array[start … p-1] are less than pivot = array [p]
                                                      // and all array[p+1 … end] are >= pivot
        quickSort (array, start, pivot_index -1);     // sort first partition of the array (from start to
                                                      // pivot_index-1)
        quickSort (array, pivot_index +1, end);       //sort second partition of the array
  else
        return              // do nothing, the array has one element, so it is sorted
}
```

Patition– Pseudo-code

```
partition (arr[], low, high) {
        // pivot (Element to be placed at right position)
        pivot = arr[high];

        i = (low - 1)  // Index of smaller element

        for (j = low; j <= high- 1; j++) {
                // If current element is smaller than the pivot
                if (arr[j] < pivot) {
                        i++;    // increment index of smaller element
                        swap arr[i] and arr[j]
                }
        }
        swap arr[i + 1] and arr[high])
        return (i + 1)
}
```

Another Way – Following the same principle

```
quickSort ( array) {
  if array.length > 1
        p = middle element
        [L, E, G] = partition (array, p);

        L = quickSort (L);   // sort first partition of the array (from start to
                             // pivot_index-1)
        G = quickSort (G);        //sort second partition of the array

        return L ∪ E ∪ G
 else
        return     array        // do nothing, the array has one element, so it is sorted
}
```

Another Way – Following the same principle

Algorithm ***partition***(***S***, ***p***)
  Input sequence ***S***, position ***p*** of pivot
  L, ***E***, ***G*** ← empty sequences
  x = ***S.remove***(***p***)
  while !***S.isEmpty***()
    y = ***S.remove***(***S.first***())
    if ***y*** < ***x***
      L.addLast(***y***)
    else if ***y*** = ***x***
      E.addLast(***y***)
    else { ***y*** > ***x*** }
      G.addLast(***y***)
  Output subsequences ***L***, ***E***, ***G*** of the elements of ***S*** less than, equal to, or greater than the pivot, resp.

- We partition an input sequence as follows:
  - We remove, in turn, each element ***y*** from ***S*** and
  - We insert ***y*** into ***L***, ***E*** or ***G***, depending on the result of the comparison with the pivot ***x***
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Partitioning Mechanism

- The Quick Sort algorithm is fastest when the median value of the array is chosen as the pivot element.
  - This occurs because the resulting partitions are of similar size. Each partition splits itself into two and thus the base case is reached very quickly.
- HOWEVER, Quick Sort algorithm becomes very slow when the array is already or close to being sorted.
  - There is no efficient way for the computer to find the median element to use as the pivot.
- So, the more random the arrangement of the array, the faster the Quick Sort algorithm finishes.
- The runtime of Quick Sort ranges from $O(N \log N)$ with the best pivots, to $O(N^2)$ with the worst pivots.

Partitioning Mechanism

- **Recipe 1: Select the pivot as the median value (prominent)**
  - How efficiently can we compute the median of an array of numbers?
  - There is no obvious way to do this quickly => it requires you to *sort* the array!
    - we are trying to use the median to do the sorting, so we can't sort the list in order to compute the median!
  - e.g., { 56, 29, 35, 42, 15,41,75,21} => {15,21,29, 35 ,41,42,56,75}
  - 35 is the median value
  - So, it is difficult to determine the median value!

Partitioning Mechanism

- **Recipe 2: Select first element as the pivot**
  - When the input array is random, ok
  - When the input array is presorted (or in reverse order)
    - The pivot is the smallest element
    - All the elements go into Sub_array2 or Sub_array1. There is no element smaller than the pivot to be placed in the other Sub_array
    - This happens consistently throughout the recursive calls
    - Results in $O(N^2)$ behavior –> worst situation

Partitioning Mechanism

- **Recipe 3: Evaluation of median based on three numbers**
  - take a sample of three elements from the array, then use the median of the three as pivot element
    - Quickest way: take the first, middle and last elements
    - Selecting the median involves sorting only 3 numbers
    - Re-arrange (sort) these three numbers so that the smallest is in the first position, the highest in the last position, and the other in the middle
  - **Advantages:**
    - it makes the worst case (pivot = smallest/ highest number) much more unlikely to occur in any actual sort
    - Speeds up the partition loop

Partitioning Mechanism

- **Recipe 4: Select the random value as the pivot**
  - generally safe
  - But random number generation can be expensive (in terms of speed)

That’s all folks!

- Any question?