Randomized Quick Sort

- **Quick Sort** is a three-step divide-and-conquer process for sorting a typical subarray A[p…..r]:

  Divide: Partition (rearrange) the array A[p….r] into two (possibly empty) subarrays A[p…q-1] and A[q+1……r] such that each element of A[p…..q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1……r]. Compute the index q as part of this partitioning procedure.

  Conquer: Sort the two subarrays A[p…..q-1] and A[q+1……r] by recursive calls to quicksort.

  Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p….r] is now sorted.

  The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

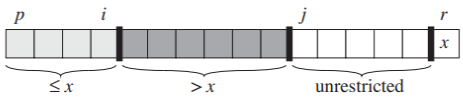
To sort an entire array A, the initial call is QUICKSORT(A, 1, $A.length$).

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p\,..\,r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

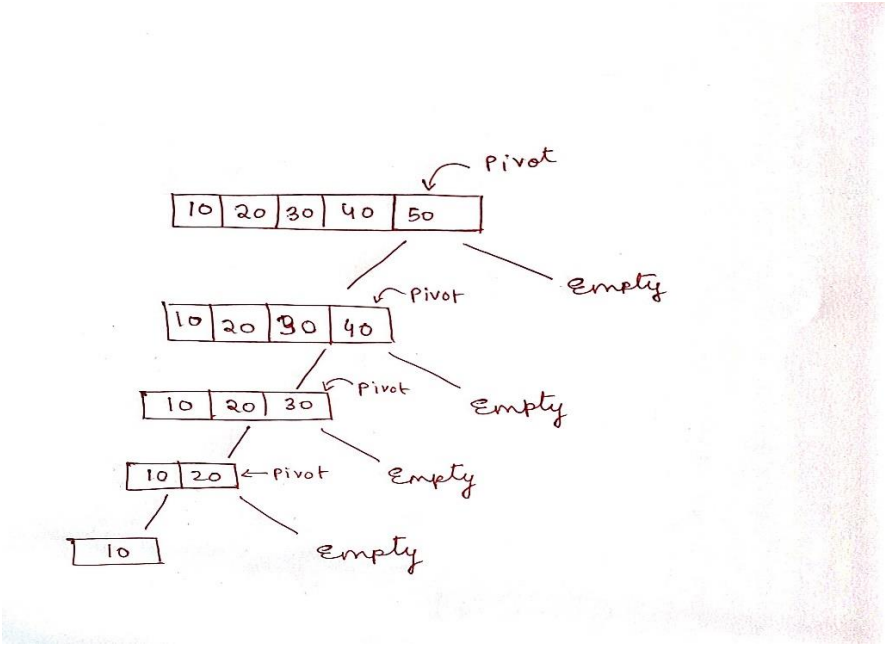
The four regions are maintained by the procedure PARTITION on a subarray A[p…..r]. The values in A[p….i] are all less than or equal to x, the values in A[i+1….j-1] are all greater than x, and A[r]=x. The subarray A[j…..r-1] can take on any values.



The final two lines of PARTITION finish up by swapping the pivot element with the leftmost element greater than x, thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot's new index

In this version of Quick Sort where the rightmost element is chosen as a pivot, the worst occurs in the following cases.
1) Array is already sorted in the same order.
2) Array is already sorted in reverse order.

It is so because in these cases, the PARTITION routine produces one subarray with n-1 elements and the other subarray with zero element, as shown in the example:



[Assuming unbalanced partitioning exists in each recursive call]

Number of comparisons= (n-1) + (n-2) + ……+ 1
= [n(n-1)]/2
So, time complexity = $O(n^2)$

Hence, we see that in the case of quick sort, deterministic selection of the pivot element can often lead to worst case.
So, we need to **randomize** the algorithm.

- **Randomized Quick Sort:**

  In randomized Quick Sort, instead of always using A[r] as the pivot, we will select a randomly chosen element from the subarray A[p……r]. We do so by first exchanging element A[r] with an element chosen at random from A[p….r]. By randomly sampling the range p,……, r, we ensure that the pivot element x =A[r] is equally likely to be any of the r – p + 1 elements in the subarray.

  The changes to PARTITION and QUICKSORT are small. In the new partition procedure, we simply implement the swap before actually partitioning:

RANDOMIZED-PARTITION(A, p, r)

```
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

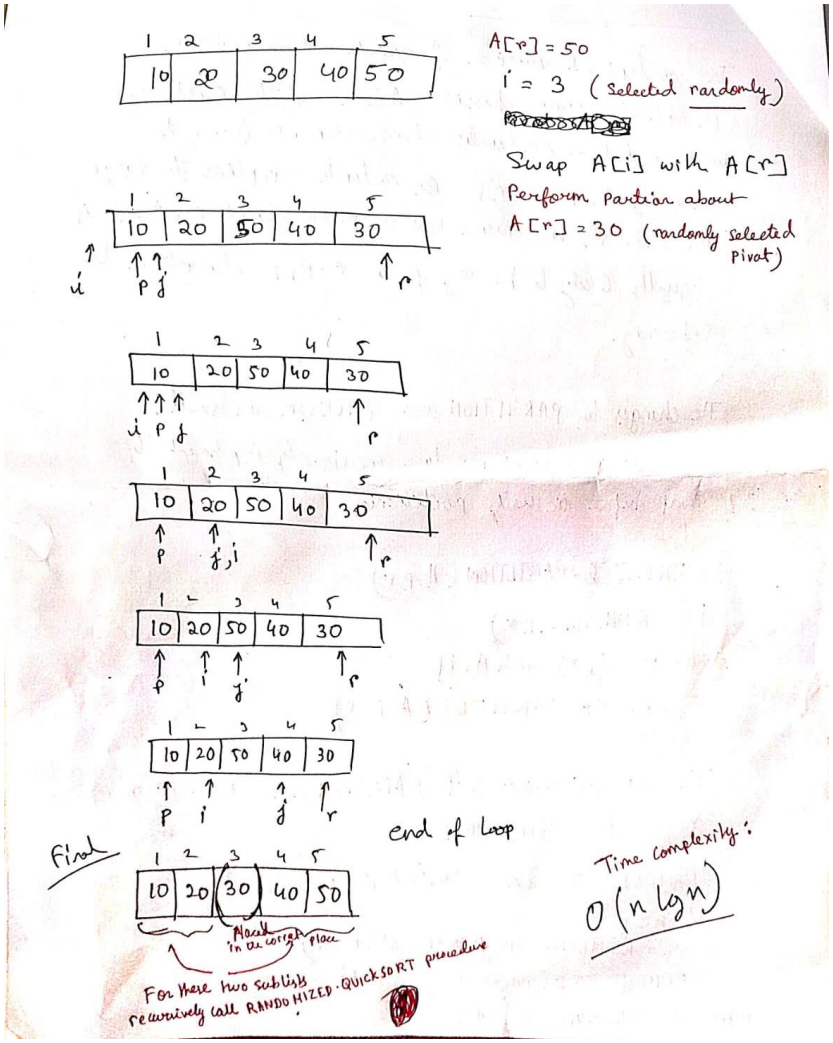
The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

```
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Example:

We will take the same array as taken, while illustrating the deterministic version of Quick Sort, and see the difference:



After the 1st Pass, A[r] is placed in the correct position.
For both the sublists, RANDOMIZED_QUICKSORT procedure is called recursively.

Hence, we see that for the same set of input data, in case of randomized version, time complexity= O(n log n), which is better than $O(n^2)$ [in case of deterministic counter part of the algorithm].

Analysis:

- **Worst Case Analysis:**
  Worst-case split at every level of recursion in quicksort produces a $O(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm. We now prove this assertion.

  Let T(n) be the worst-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \tag{7.1}$$

where the parameter q ranges from 0 to $n-1$ because the procedure PARTITION produces two subproblems with total size $n-1$. We guess that $T(n) \le cn^2$ for some constant c. Substituting this guess into recurrence (7.1), we obtain

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

The expression $q^2 + (n-q-1)^2$ achieves a maximum over the parameter's range $0 \le q \le n-1$ at either endpoint. To verify this claim, note that the second derivative of the expression with respect to q is positive (see Exercise 7.4-3). This observation gives us the bound $\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuing with our bounding of $T(n)$, we obtain

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

since we can pick the constant c large enough so that the c(2n-1) term dominates the O(n) term. Thus, $T(n) = O(n^2)$

- **Average Case:**

Lemma 7.1
Let X be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

Proof By the discussion above, the algorithm makes at most n calls to PARTITION, each of which does a constant amount of work and then executes the **for** loop some number of times. Each iteration of the **for** loop executes line 4. ■

Our goal, therefore, is to compute X, the total number of comparisons performed in all calls to PARTITION. We will not attempt to analyze how many comparisons are made in *each* call to PARTITION. Rather, we will derive an overall bound on the total number of comparisons. To do so, we must understand when the algorithm compares two elements of the array and when it does not. For ease of analysis, we rename the elements of the array A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element. We also define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ to be the set of elements between z_i and z_j, inclusive.

When does the algorithm compare z_i and z_j? To answer this question, we first observe that each pair of elements is compared at most once. Why? Elements are compared only to the pivot element and, after a particular call of PARTITION finishes, the pivot element used in that call is never again compared to any other elements.

Our analysis uses indicator random variables (see Section 5.2). We define

$$X_{ij} = \mathrm{I}\{z_i \text{ is compared to } z_j\} ,$$

where we are considering whether the comparison takes place at any time during the execution of the algorithm, not just during one iteration or one call of PARTITION. Since each pair is compared at most once, we can easily characterize the total number of comparisons performed by the algorithm:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

Taking expectations of both sides, and then using linearity of expectation and Lemma 5.1, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} .
\end{aligned}$$

We now compute the probability that this event occurs. Prior to the point at which an element from Z_{ij} has been chosen as a pivot, the whole set Z_{ij} is together in the same partition. Therefore, any element of Z_{ij} is equally likely to be the first one chosen as a pivot. Because the set Z_{ij} has $j-i+1$ elements, and because pivots are chosen randomly and independently, the probability that any given element is the first one chosen as a pivot is $1/(j-i+1)$. Thus, we have

$$\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} . \qquad (7.3)
\end{aligned}$$

The second line follows because the two events are mutually exclusive. Combining equations (7.2) and (7.3), we get that

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

We can evaluate this sum using a change of variables ($k = j - i$) and the bound on the harmonic series in equation (A.7):

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) . \qquad (7.4)
\end{aligned}$$

Thus we conclude that, using RANDOMIZED-PARTITION, the expected running time of quicksort is $O(n \lg n)$ when element values are distinct.