classmate Date Page

QuickSort

- Quick Sort algorithm uses divide and conquer paradigm.

- Divide - Partition the array $A[p..r]$ into two 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 $A[q]$, each element of $A[q+1...r]$ is greater than or equal $A[q]$
  - Compute index q for partitioning.

- Conquer - Sort above Two Subarrays $A[p..q-1]$ and $A[q+1...r]$ by recursive Call to quickSort.

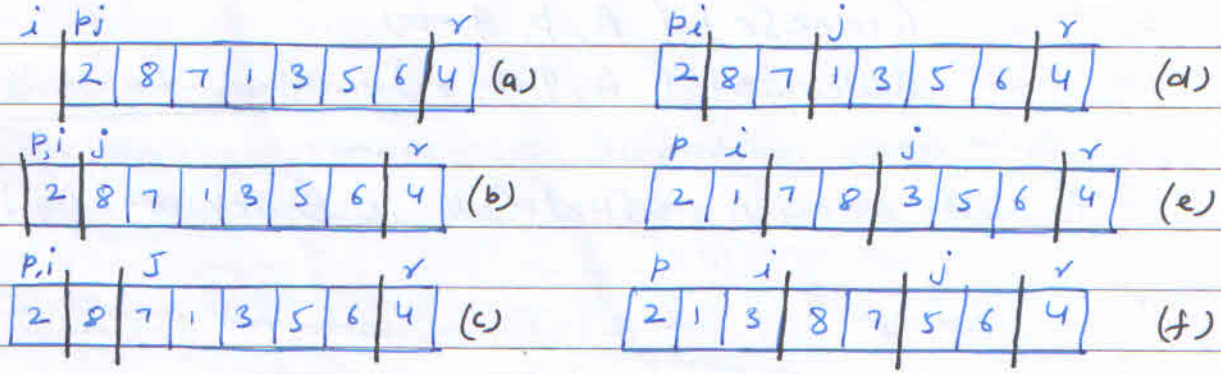
- Combine - As Subarrays are already sorted, no work is needed to Combine, entire array $A[p..r]$ is now Sorted.

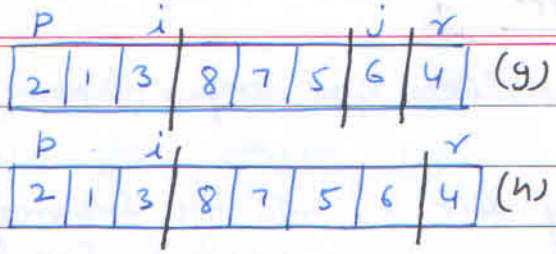
Partitioning the Array

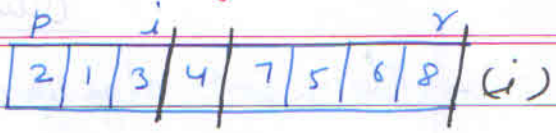
PARTITION (A, p, r)

```
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
```

Example

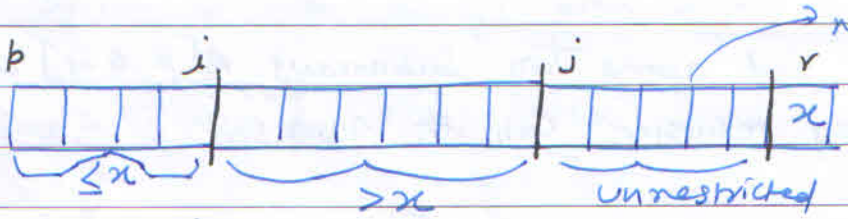






- Partition procedure always select an element $x = A[r]$ as pivot

- Procedure partition the array into 4 regions.



- Each iteration of for loop region satisfy certain properties These properties are as a loop invariant
- At the beging of each iteration of the loop for any array index K.
  1. If $p \le k \le i$, then $A[k] \le x$
  2. If $i+1 \le k \le j-1$ then $A[k] > x$
  3. If $k = r$ then $A[k] = x$.

- Running time of Partition is $\Theta(n)$ where $n = r - p + 1$.

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 array A, initial Call is QuickSort(A, 1, A.length)

Performance of Quicksort

- Running time depends on whether the partitioning is balanced or unbalanced.
- If balanced, algo runs asymptotically as fast as merge sort
- If unbalanced, it run asymptotically as slow as Insertion sort.

Worst Case Partitioning

- When partitioning, routine produces one subproblem with $n-1$ elements and one with 0 elements
- Assume this unbalanced partitioning arises in each recursive call
- Partitioning cost $\Theta(n)$,
- $T(0) = \Theta(1)$ (array of size 0)
- recurrence relation - $T(n) = T(n-1) + T(0) + \Theta(n)$

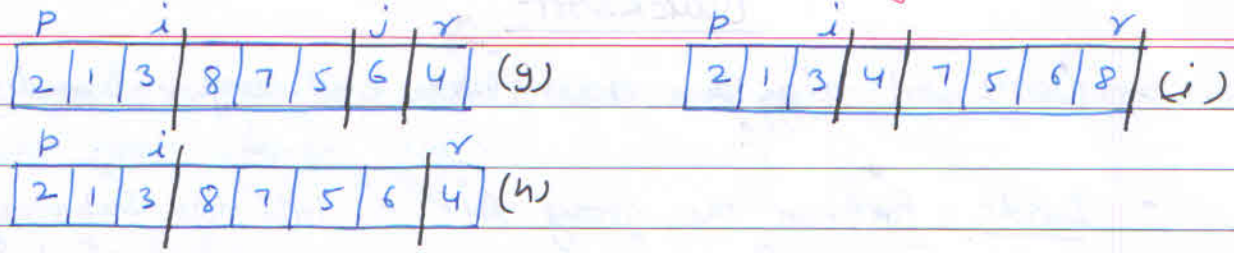
$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

- $\Theta(n^2)$ running occur when array is already sorted. In similar situation Insertion Sort runs in $O(n)$ time.

Best Case - partitioning

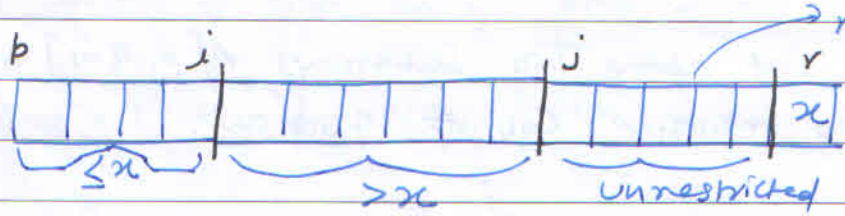
- Partitioning produce two subproblem each of size no more than $n/2$, one is $n/2$ other is $n/2 - 1$.
- Recurrence relation - $T(n) = 2T(n/2) + \Theta(n)$

$$T(n) = \Theta(n \log n)$$



- Partition procedure always select an element $x = A[r]$ as pivot

- Procedure partition the array into 4 regions.



- Each iteration of for loop region satisfy certain properties These properties are as a loop invariant
- At the beging of each iteration of the loop for any array index k.
  1. If $p \le k \le i$, then $A[k] \le x$
  2. If $i+1 \le k \le j-1$ then $A[k] > x$
  3. If $k = r$ then $A[k] = x$.

- Running time of Partition is $\Theta(n)$ where $n = r-p+1$.

QuickSort(A, p, r)

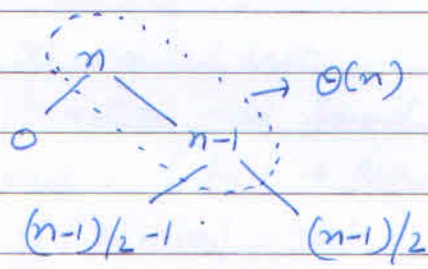
```
1. If p < r
2.     q = PARTITION(A, p, r)
3.     QuickSort(A, p, q-1)
4.     QuickSort(A, q+1, r)
```

- To sort array A, initial Call is QuickSort(A, 1, A.length)

Quick Sort

Average Case

- Partition produce a mix of 'good' and 'bad' splits.
- In recursion tree good and bad splits are distributed randomly throughout the tree.
- Good and bad splits alternate level occur in the tree
  (BestCase) (WorstCase)



- Combination of split produce 3 sub arrays of size 0, $(n-1)/2 - 1$ and $(n-1)/2$ at cost $\Theta(n) + \Theta(n-1) = \Theta(n)$
  (goodSplit + badSplit)
- Running time of average Case is $O(n \log n)$ like the running time of best Case (good splits) but slightly larger constant hidden by the O-notation.

Ques. Running time of Quicksort when all the elements have same value.
Ans. $\Theta(n^2)$, one partition is always empty (worst Case)

Binary Search

Searching

- A Simple appriach is to do linear Search whose Complexity is $O(n)$.

- The idea of binary Search is to use the information that an array is Sorted and reduce the time Complexity to $O(\log n)$.

BinarySearch(A, val, left, Rgut) (Recursive approach)

```
1.  If rigut < left
2.     return not found
3.  mid = (left + Rigut)/2
4.  if (A[mid] > val)
5.        Return BinarySearch(A, val, left, mid-1)
6.  Else If (A[mid] < val)
7.        Return BinarySearch(A, val, mid+1, Rigut)
8.  Else
9.     Return mid
```

- In this we ignore half of the elements just after one Comparision.

1. Compare value (val) with middle element.
2. if val matches with middle element we return middle index.
3. Else if val is greater than mid element, the value Can only lie in right half Subarray after mid element. So we recur for Right half.
4. Else (val is Smaller) recur for left half.

- Recursive Binary Search algorithm uses Divide and Conquer Strategy.

- Recurrence relation
  $T(n) = T(n/2) + \Theta(1)$
  $T(n) = \Theta(\log n)$ for worst case running time.

- Auxilary space - $O(1)$ for iterative implementation.
  - $O(\log n)$ recursion Call Stack Space.

- Algorithmic Paradigm - Divide and Conquer.

Binary Search_iterative (A, l, r, x)

```
1. While (l <= r)
2.       m = (l + r)/2
3.       if (A[m] == x) return m;
4.       if (A[m] < x) l = m+1;
5.       Else r = m-1
6. End While.
```

Ques Why Binary Search is better than Ternary Search?