Sorting

Sorting is the process of arranging a set of data in some order. Two types of sorting:

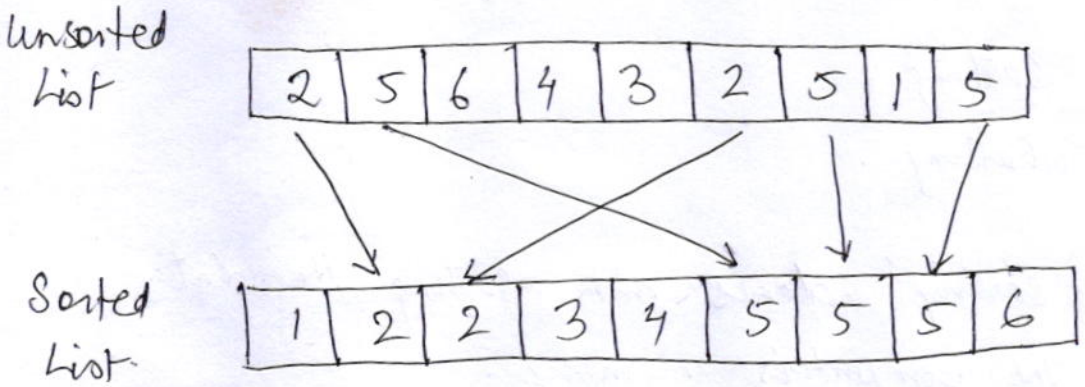
1. Internal sorting
2. External sorting.

1. Internal sorting deals with sorting the data stored in the computer's memory.
2. External sorting is required when the data to be sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single large file.

Stable Sort

A list of unsorted data may contain two or

more equal data. If a sorting method maintains the same relative position of their occurrences in the sorted list, then it is called stable sort.



Stable sort.

Complexity of Algorithm.

To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Frequently the storage space required by an algorithm is simply a multiple of the data size n. Accordingly, unless otherwise stated, the term 'complexity' shall refer to the running time of the algorithm. Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run, as a function

of the size of the input to the problem. The time complexity of an algorithm is commonly expressed using the big 'O' notation, which suppresses multiplicative ~~cont~~ constants and lower order terms. When expressed this way, the time complexity is said to be described asymptotically, i.e, as the input size goes to infinity. For example: if the time required by an algorithm on all input of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$. Time complexity is commonly estimated by counting the number of key operations performed by the algorithm - in sorting and searching algorithms, for example, the number of comparisons.

(*) The complexity of an algorithm M is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multip

of the data size n. Accordingly, unless otherwise stated, the term complexity shall refer to the running time of the algorithm.

Clearly the complexity $f(n)$ of n increases as n increases. It is usually the rate of increase of $f(n)$ that we want to examine. This is usually done by comparing $f(n)$ with some standard functions such as: $\log_2 n$, n, $n \log_2 n$, n^2, n^3, 2^n.

The functions are listed in the order of their rates of growth: the logarithmic function $\log_2 n$ grows most slowly, the exponential function 2^n grows most rapidly, and the polynomial function n^c grow according to the exponent c.

One way to compare the function $f(n)$ with these standard functions is to use the asymptotic notations (O, Ω, Θ). Expressing the complexity function with reference to other known functions is called asymptotic complexity.

Bubble Sort

BUBBLE (A; N)

Complexity is $O(n^2)$

Here A is an array with N elements.

1. Repeat for I = 1 to N-1
2. Repeat for J = 1 to N-I
3. If A[J] > A[J+1] then.
    1. TEMP = A[J]
    2. A[J] = A[J+1]
    3. A[J+1] = TEMP.
4. End if
5. End for
6. End for.
7. Stop.

Suppose the list of elements A[1], A[2], ..., A[N]. is in memory, the bubble sort algorithm works as follows:

Step 1: Compare A[1] and A[2] and arrange them in the desired order, so that A[1] < A[2]. Then compare A[2] and A[3] and arrange them so

that $A[2] < A[3]$. Continue until we compare $A[N-1]$ with $A[N]$ and arrange them so that $A[N-1] < A[N]$.

(During Step 1, the largest element is 'bubbled up' to the N^{th} position) When step 1 is ~~Complete~~ completed $A[N]$ will contain the largest element.

Step 2. Repeat step 1 with one less comparison so that now we stop after we compare and possibly rearrange $A[N-2]$ and $A[N-1]$.

⋮

Step $N-1$. Compare $A[1]$ and $A[2]$ and arrange them so that $A[1] < A[2]$.

Example:

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57.

Pass 1. We have the following comparisons.

I. Compare A1 and A2, Since 32 < 51, the list is same.

II. Compare A2 and A3. Since 51 > 27 interchange 51 and 27 as follows:

32, (27), (51), 85, 66, 23, 13, 57

III. Compare A3 and A4. Since 51 < 85, the list is not changed.

IV. Compare A4 and A5. Since 85 > 66, Interchange 85 and 66 as follows:

32, 27, 51, (66), (85), 23, 13, 57

V. Compare A5 and A6, 85 > 23, Interchange 85 and 23.

32, 27, 51, 66, (23), (85), 13, 57

VI. Compare A6 and A7, 85 > 13, interchange 85 and 13

32, 27, 51, 66, 23, (13), (85), 57

VII. Compare A7 and A8, 85 > 57, interchange

32, 27, 51, 66, 23, 13, (57), (85)

Pass 2: (27), (33), 51, 66, 23, 13, 52, 85

27 33 51 (23) (66) 13 57 85

27 33 51 23 (13) (66) 57 85

27 33 51 23 13 (57) (66) 85

At the end of pass 2, the second and largest number 66 has moved to the 2nd last position.

Pass 3 27, 33, (23), (51), 13, 57, 66, 85

27, 33, 23, (13), (51), 57, 66, 85

Pass 4

27, (23), (33), 13, 51, 57, 66, 85

27, 23, (13), (33), 51, 57, 66, 85

Pass 5

(23), (27), 13, 33, 51, 57, 66, 85

23, (13), (27), 33, 51, 57, 66, 85

Pass 6

(13), (23), 27, 33, 51, 57, 66, 85

Pass 7

Finally A[1] is compared with A[2]. Here no interchange takes place as 13 < 23.

⊛ Complexity of Bubble Sort Algorithm.

Traditionally the time for a sorting algorithm is measured in terms of the number of ~~compars~~ comparisons. The number $f(n)$ of comparisons in the bubble sort is easily computed. Specifically, there are $(n-1)$ comparisons during the first pass, which places the largest element in the last position. There are $(n-2)$ comparisons in the second pass which places the second largest element in the next to last position and so on. Thus.

$$f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

$$= n^2/2 + O(n) = \theta(n^2).$$

In other words, the time required to execute the bubble sort algorithm is proportional to n^2 where n is the number of input items.

Selection Sort

SELECTION (A, N).

Here A is a linear array with N elements

1. Repeat for I = 1 to N − 1.
    1. Set MIN = A[I].
    2. Set LOC = I
    3. Repeat for J = I + 1 to N
        1. If A[J] < MIN then
            1. Set MIN = A[J]
            2. Set LOC = J
        2. Endif
    4. End for.
    5. TEMP = A[I]
    6. A[I] = A[LOC]
    7. A[LOC] = TEMP
2. End for.
3. Stop.

Suppose an array A with N elements A[1], A[2], ..., A[N] is in memory. The selection sort algorithm for sorting A works as follows.

First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on.

More precisely:

Pass 1: Find the location LOC of the smallest in the list of N elements. A[1], A[2], ... A[N]. and then interchange A[LOC] and A[1]. Then A[1] is sorted.

Pass 2: Find the location LOC of the smallest in the sublist of N-1 elements, A[2], A[3], ..., A[N] and then interchange A[LOC] and A[2]. Then A[1], A[2] is sorted, since $A[1] \leq A[2]$.

⋮

Pass N-1: Find the location LOC of the smaller of the elements A[N-1] and A[N] and then interchange A[LOC] and A[N-1] Then A[1], A[2], ..., A[N-1], A[N] is sorted after N-1 passes.

Example:

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55.

Applying the selection sort algorithm to A yields the data in the fig. ~~below~~ below. Observe that LOC gives the location of the smallest among A[K], A[K+1], ..., A[N] during Pass k. The circled elements indicate the elements which are to be interchanged.

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
I=1, LOC=4	(77)	33	44	(11)	88	22	66	55
I=2, LOC=6	11	(33)	44	77	88	(22)	66	55
I=3, LOC=6	11	22	(44)	77	88	(33)	66	55
I=4, LOC=6	11	22	33	(77)	88	(44)	66	55
I=5, LOC=8	11	22	33	44	(88)	77	66	(55)
I=6, LOC=7	11	22	33	44	55	(77)	(66)	88
I=7, LOC=7	11	22	33	44	55	66	(77)	88

Sorted list: 11, 22, 33, 44, 55, 66, 77, 88

(*) There remains only the problem of finding, during the k^{th} pass, the location LOC of the smallest among the elements $A[k], A[k+1], \ldots, A[N]$. This may be accomplished by using a variable MIN to hold the current smallest value while scanning the sublist from $A[k]$ to $A[N]$.

(*) Specifically, first set $MIN = A[k]$ and $LOC = k$ and then traverse the list, comparing MIN with each other element $A[J]$ as follows:

a) If $MIN \leq A[J]$ then simply move to the next element.

b) If $MIN > A[J]$ then update MIN and LOC by setting $MIN = A[J]$ and $LOC = J$.

(*) After comparing MIN with the last element $A[N]$, MIN will contain the smallest among the elements $A[k], A[k+1], \ldots, A[N]$ and LOC will contain its location.

(*) Complexity of Selection Sort.

The ~~number~~ number $f(n)$ of comparisons in the selection sort is independent of the original order of the elements. Observe that $MIN(A, k, N, LOC)$ requires $(n-k)$ comparisons. That is, there are

$n-1$ comparisons during pass 1 to find the smallest element. There are $n-2$ comparisons during pass 2 to find the second smallest element and so on. Accordingly $f(n) = (n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2 = O(n^2)$.

Insertion Sort.

Here A is an array with N elements.

```
INSERTION (A, N).
1. Repeat for I = 2 to N
       Set
   1. key = A[I]
   2. Set J = I - 1
   3. Repeat while (J >= 1 and key < A[J])
      1. Set A[J+1] = A[J] // Move forward one location
      2. Set J = J - 1
   4. End while
   5. Set A[J+1] = key.
2. End for.
3. Stop
```

Suppose an array A with N elements A[1], A[2], ..., A[N] is in memory. The insertion sort

algorithm scans A from A[1] to A[N], inserting each element A[k] into its proper position in the previously sorted subarray A[1], A[2], ..., A[k-1]. That is:

Pass 1: A[1] by itself is trivially sorted.

Pass 2: A[2] is inserted either before or after A[1] so that A[1], A[2] is sorted.

Pass 3: A[3] is inserted into its proper place in A[1], A[2], that is, before A[1], between A[1] and A[2], or after A[2], so that A[1], A[2], A[3] is sorted.

⋮

Pass N: A[N] is inserted into its proper place in A[1], A[2], ..., A[N-1] so that A[1], A[2], ..., A[N] is sorted.

This sorting algorithm is frequently used when n is small. There remains only the problem of deciding how to insert A[k] in its proper place in the sorted subarray A[1], A[2], ..., A[k-1]. This can be accomplished by comparing

A[K] with A[k-1], comparing A[k] with A[k-2], comparing A[k] with A[k-3] and so on, until first meeting an element A[J] such that $A[J] \leq A[k]$. Then each of the elements A[k-1], A[k-2], ..., A[J+1] is moved forward one location, and A[k] is then inserted in the (J+1)st position in the array.

Example:

Suppose an array A contains 8 elements as follows: 77, 33, 44, 11, 88, 22, 66, 55.

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
k=1	77	33	44	11	88	22	66	55
k=2	77	(33)	44	11	88	22	66	55
k=3	33	77	(44)	11	88	22	66	55
k=4	33	44	77	(11)	88	22	66	55
k=5	11	33	44	77	(88)	22	66	55
k=6	11	33	44	77	88	(22)	66	55
k=7	11	22	33	44	77	88	(66)	55
k=8	11	22	33	44	66	77	88	(55)

Sorted list :

11, 22, 33, 44, 55, 66, 77, 88

The circled element indicates the A[k] in each pass of the algorithm and the arrow indicates the proper place for inserting A[k].

⊛ Complexity of Insertion Sort

- To insert last or N^{th} element we need at most $N-1$ comparisions and movement.
- To insert $(N-1)$ element we need at most $N-2$ comparisons and movement. and so on.
- To insert 2nd element we need at most 1 comparison and movement

To sum up:

$$2 * ((N-1) + (N-2) + \cdots + 2 + 1) = 2(n * (n-1)/2)$$

$$= O(n^2).$$

Complexity of Insertion sort is $O(n^2)$.

Merge Sort Algorithm.

The mergesort technique follows the divide and conquer paradigm.

Let A be a list of N elements to be sorted with LOW and HIGH being the position of leftmost and rightmost elements in the list. The divide and conquer technique has 3 tasks namely:

1. Divide: Partitions the list midway that is at $(LOW + HIGH)/2$ into two sublists with $N/2$ elements in each if N is even or $N/2 - 1$ elements if N is odd. The task divide merely calculates the middle index of the list.

2. Conquer: Sorts the two lists recursively using the mergesort. The conquer step is to solve the same problem but for two smaller lists. It can be done recursively and the recursion should terminate when the list to be sorted has length 1, in which case, every list of length 1 is already in sorted order.

3. Combine : Merge the sorted sublists to obtain the sorted output. The combine step merges two adjacent subarrays and put the resulting merged array back into the same storage space originally occupied by the elements being merged.

. MERGESORT (A, LOW, HIGH).

A is an array with LOW and HIGH as the lower and upper index of A.

1. If (LOW < HIGH)
    1. Set MID = INT (LOW + HIGH)/2 // Divide
    2. Call MERGESORT (LOW, MID) } // Conquer left
    3. Call MERGESORT (MID+1, HIGH) } and right sublists
    4. Call MERGE (LOW, MID, HIGH) // Combine
2. End if
3. Stop.

The MERGESORT algorithm is defined recursively.

MERGE (LOW, MID, HIGH)

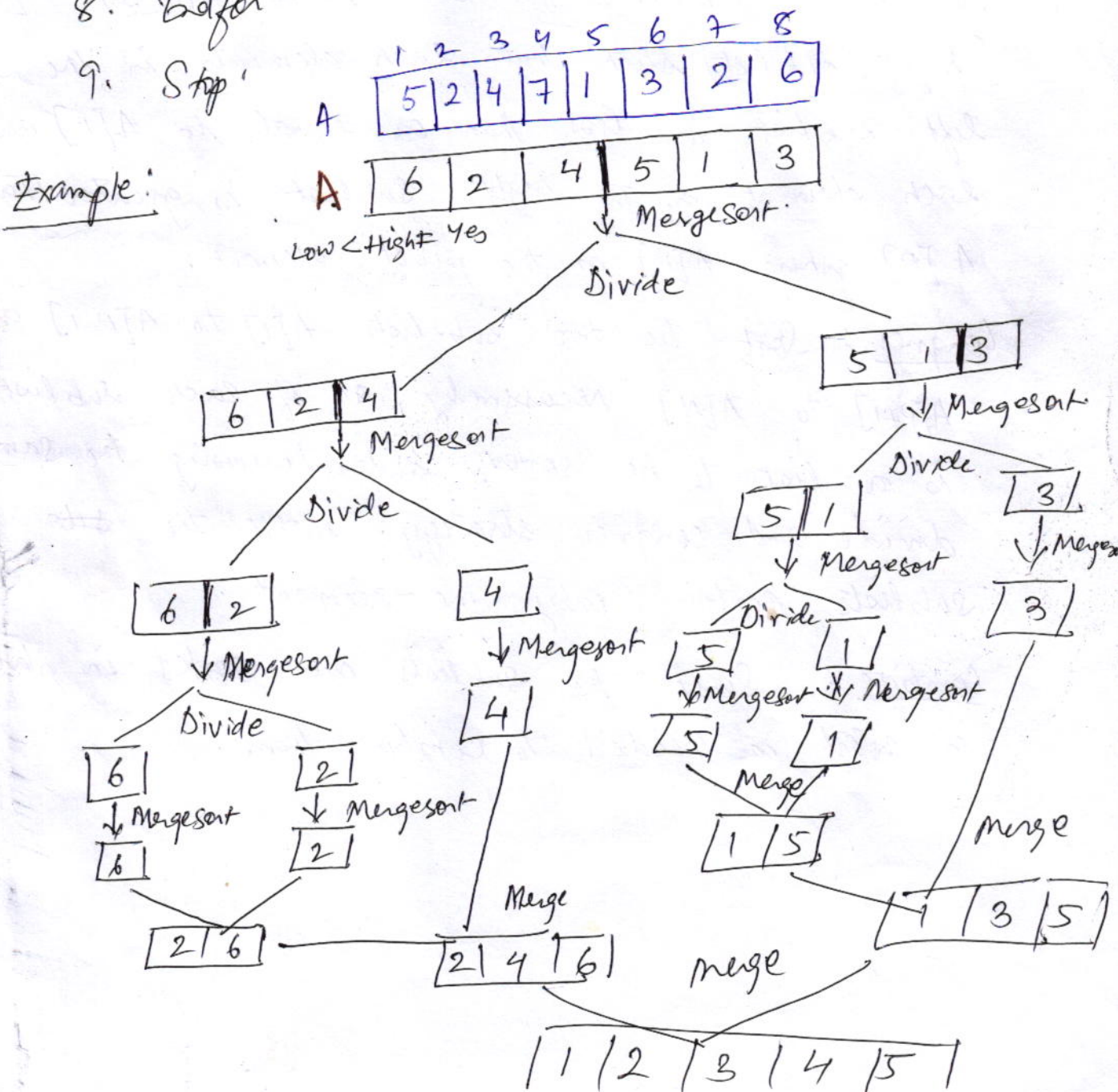
The input is an array A with two subarrays where indices range from LOW to MID and MID+1 to HIGH

// I and J point at the beginning of the two subarrays and k to B.

1. Set I = LOW, J = MID+1, K = LOW // 1.
2. Repeat while (I ≤ MID and J ≤ HIGH)
    1. If (A[I] ≤ A[J]) then.
        1. B[k] = A[I].
        2. I = I+1
    2. Else
        1. B[k] = A[J]
        2. J = J+1
    3. Endif.
    4. k = k+1
3. Endwhile.
4. If (I > MID) then // copy rest of right subarray to B.
    1. Repeat for L = J to HIGH.
        1. B[k] = A[L].
        2. k = k+1
    2. Endfor.

5. Else // Copy rest of left subarray to B.
   1. Repeat for L = I to MID.
      1. B[k] = A[L]
      2. K = k+1
   2. Endfor.
6. Endif
7. Repeat for I = LOW to HIGH // Copy B to A.
   1. A[I] = B[I].
8. Endfor
9. Stop.

Example:



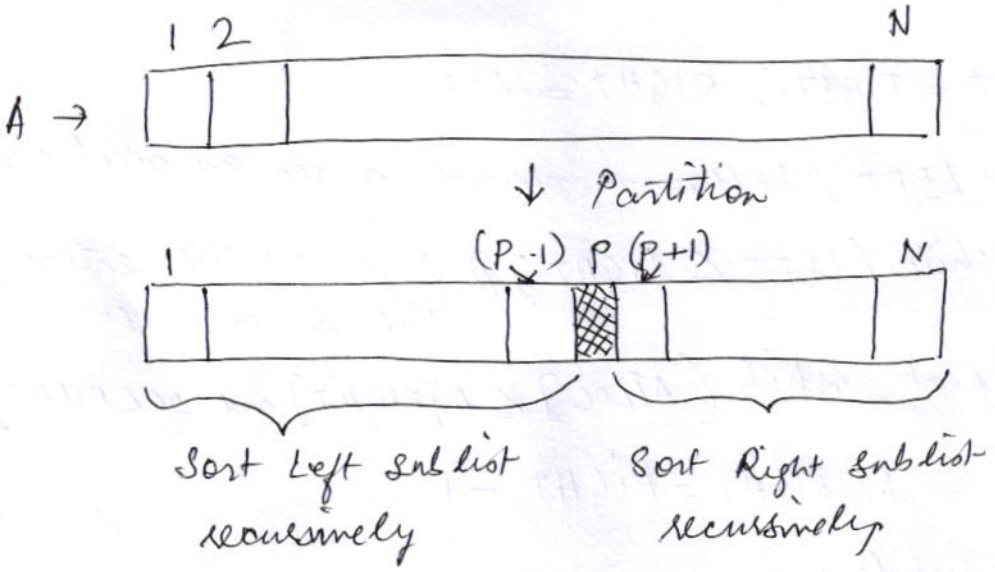
Quicksort Algorithm.

The quick sort algorithm uses the divide and conquer strategy, dividing a large list into a number of smaller lists, sorts them seperately and then combine the results to get the original list sorted. Let A be a list of N elements to be sorted.

Divide: Partition the list A into two sublists A[1], A[2], ..., A[p-1] (left sublist) and A[p+1], ..., A[N] (right sublist) such that each elements in the left sublist is less than or equal to A[p] and each element in the right sublist is greater than A[p] where A[p] is the pivot element.

Conquer: Sort the two sub-lists A[1] to A[p-1] and A[p+1] to A[N] recursively, as if each sublist is a list to be sorted and following the same divide and conquer strategy until the ~~sub~~ sublists contain only one element.

Combine: Since the sublists are sorted in place, no work is needed to combine them.



QUICKSORT (A, FIRST, LAST)

Let A be the array with FIRST and LAST being the position of the ~~last~~ first and last elements in the list.

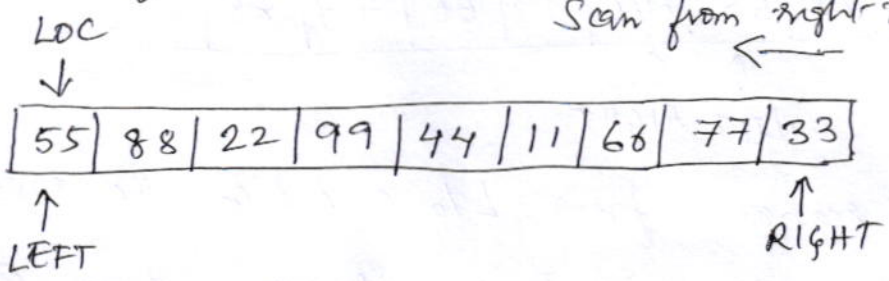
1. If (FIRST < LAST) then // Termination condition
2. P = PARTITION (FIRST, LAST)
3. QUICKSORT (FIRST, P-1) // Recursively call for the left Sublist
4. QUICKSORT (P+1, LAST) // Recursively call for the right Sublist
5. Endif
6. Stop.

PARTITION (FIRST, LAST)

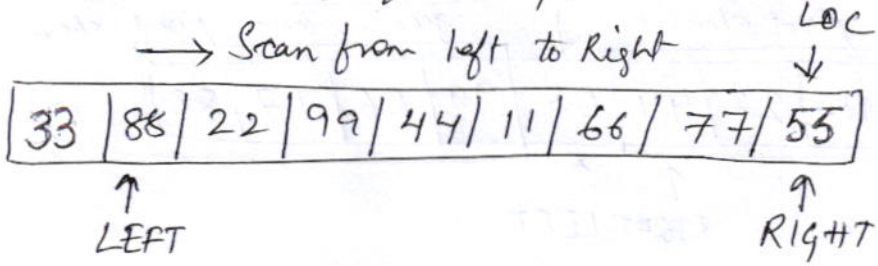
The leftmost and rightmost elements of the list under partition are located at LEFT and RIGHT, respectively. The algorithm returns LOC as the final position of the pivot element.

1. Set LEFT = FIRST , RIGHT = LAST.
2. Set LOC = LEFT // Leftmost element chosen as pivot elemat.
3. Repeat while (LEFT < RIGHT) // Repeat until entire list is scanned.
    1. Repeat while (A[LOC] ≤ A[RIGHT] and LOC<RIGHT)
        1. RIGHT = RIGHT - 1
    2. Endwhile.
    3. If (A[LOC] > A[RIGHT]) then
        1. Swap (A[LOC], A[RIGHT])
        2. LOC = RIGHT
        3. LEFT = LEFT + 1.
    4. Endif
    5. Repeat while (A[LOC] ≥ A[LEFT] and LOC > LEFT)
        1. LEFT = LEFT + 1
    6. Endwhile
    7. If (A[LOC] < A[LEFT]) then
        1. Swap (A[LOC], A[LEFT]).
        2. LOC = LEFT
        3. RIGHT = RIGHT - 1
    8. Endif.
4. Endwhile
5. Return(LOC).
6. Stop.

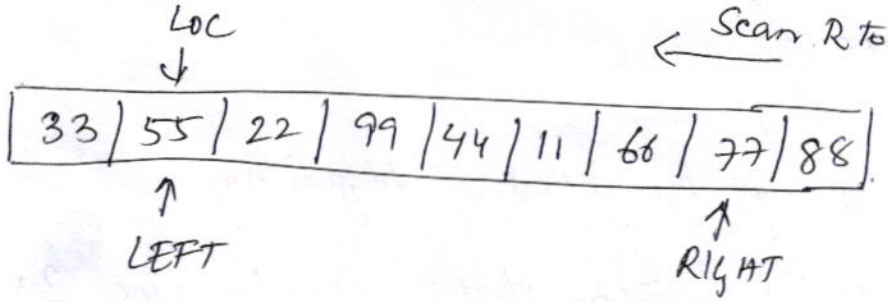
Illustration of Partition method :



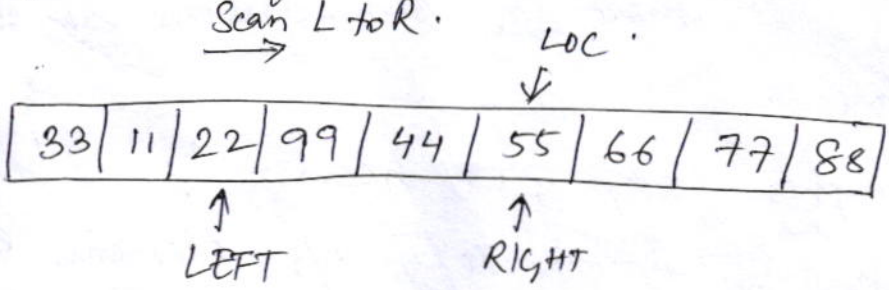
(A) At the beginning of the partition



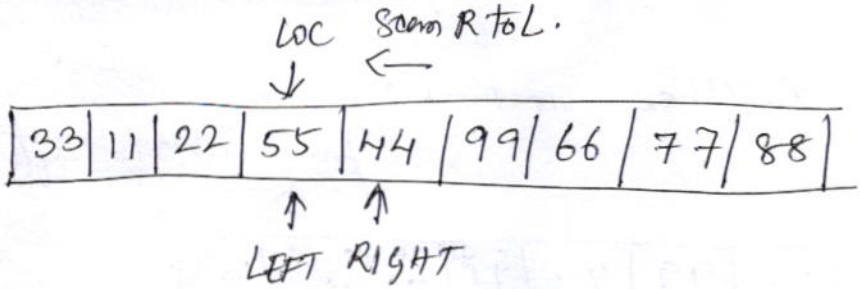
(B) After scanning from right to left while the pivot is at left. Next, it begin scanning from LEFT to RIGHT.



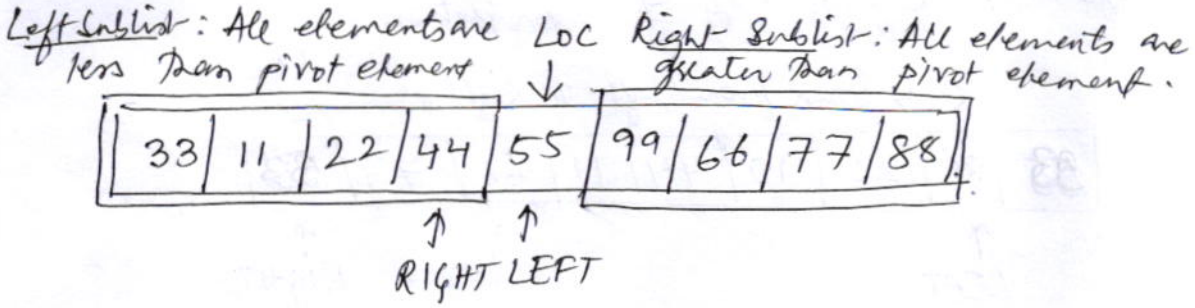
(C) After scanning from left to right, while the pivot is at right. Next it begin scanning from right to left.



(D) After scanning from R to L, while pivot is at left. Next it begin, the scanning L to R.



E) After scanning from L to R while the pivot is at RIGHT, next it begins scanning from R to L.



(F) After scanning from R to L while pivot is at left. LEFT < RIGHT condition violates. Partition is done.

⊛ Complexity of Mergesort Algorithm

1. The divide step that computes the midpoint of a subarray takes a constant time $O(1)$.

2. The combine step merges a total of n elements at each pass, requiring at most n comparisons so it takes $O(n)$ times.

3. The conquer step, we recursively sort two subarrays of approximately $n/2$ elements each. The algorithm requires approximately $\log_2 n$ passes to sort an array of size n. So the total time complexity is $O(n \log n)$.

⊛ Complexity of Quick Sort Algorithm.

Each reduction step of the algorithm produces two sublists. Accordingly.

1. Reducing the initial list places 1 element and produces 2 sublists.
2. Reducing the two sublists places 2 elements and produces 4 sublists. And so on. The reduction step in the k^{th} level finds the location of 2^{k-1} elements. Hence there will be approximately $\log n$ levels of reduction steps. Furthermore, each level uses at most n comparisons, so $f(n) = O(n \log n)$.

⊛ Complexity of Binary search.

Quicksort (A, P, R)

{ 1. If (P < R)

1. Q = Partition (A, P, R)
2. Quicksort (A, P, Q-1)
3. Quicksort (A, Q+1, R).

2. End if.

3. Stop

Partition (A, P, R)

1. $x = A[R]$
2. $I = P - 1$
3. For $J = P$ to $R - 1$.

1. If ($A[J] \leq x$) then

1. $I = I + 1$
2. SWAP ($A[I]$, $A[J]$)

2. Endif.

4. Endfor.
5. SWAP ($A[I+1]$, $A[R]$)
6. Return $I + 1$

Description of QuickSort

Quicksort is based on the divide and conquer paradigm for sorting an array $A[P..R]$.

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 $A[Q]$, which in turn is less than or equal to each element of $A[Q+1..R]$

Conquer: Sort the two subarrays $A[P..Q-1]$ and $A[Q+1..R]$ by recursive calls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them: The entire array $A[P..R]$ is now sorted.

- Internal Sorting
  External Sort.
  Stable Sort.
  Not Stable Sort – Quick Sort, HeapSort
  In place Sort Algorithm.
  Not in place Sort Algorithm.

Time Complexity.

(*) Motivation

- Measure how fast our algorithm perform
- Compare the performance of two or more algorithms.

(*) How to Find

- Time complexity is defined as a function $f(n)$ of the size of input to the problem.
- $f(n)$ compared with standard functions ($\log n, n, n^2, n^3, 2^n, \ldots$) using asymptotic notations (O, Ω, Θ)
- Example of Linear search $O(n)$ (how)
- Time complexity is estimated by counting the number of key operations performed by the algorithm.