ADITYA COLLEGE OF ENGINEERING & TECHNOLOGY

DATA STRUCTURES

UNIT I : Searching and Sorting
Part - II

Branch: I-II IT

T. Srinivasulu

Dept. of Information Technology

Aditya College of Engineering & Technology

Surampalem.

Contents

Data Structures

- Definition
- Classification of Data Structures
- Operations on Data Structures
- Abstract Data Type (ADT)
- Preliminaries of algorithms.
- Time and Space complexity.

Searching

- Linear search
- Binary search
- Fibonacci search

Sorting

- Insertion sort,
- Selection sort,
- Exchange (Bubble sort, quick sort),
- distribution (radix sort),
- merging (Merge sort) algorithms.

SEARCHING

- **Searching** means to find whether a particular value is present in an array or not.
- If the value is **present** in the array, then searching is said to be **successful** and the searching process gives the location of that value in the array.
- However, if the value is **not present** in the array, the searching process displays an appropriate message and in this case searching is said to be **unsuccessful**.
- Three popular searching methods are used.
    1. Linear search
    2. Binary search
    3. Fibonacci search

Linear search

- Linear search, also called as **sequential search**, is a very simple method used for searching an array for a particular value.
- It works by **comparing** the value to be searched with every element of the array one by one in a sequence until a match is found.
- Linear search is mostly used to search an **unordered list of elements** (array in which data elements are not sorted).
- **For example**, if an array A[] is declared and initialized as,

```
int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

- the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm for linear search

```
LINEAR_SEARCH(A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:              IF A[I] = VAL
                          SET POS = I
                          PRINT POS
                          Go to Step 6
                     [END OF IF]
                      SET I = I + 1
             [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Linear Search Algorithm.

worst case → $O(n)$

best case → $\Omega(1)$

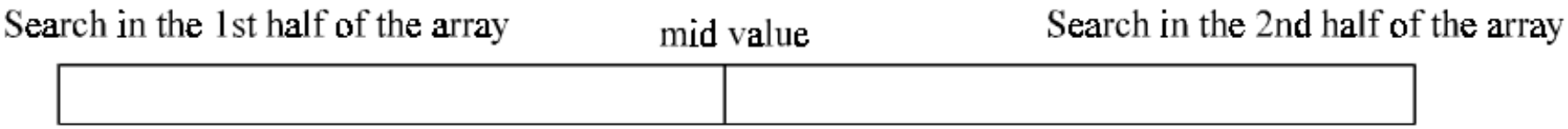
average case → $\Theta(n/2) \Rightarrow \Theta$

Exercise -1 (Searching)

a)Write C program that use both recursive and non recursive functions to perform Linear search for a Key value in a given list.

Binary search

- Binary search is a searching algorithm that works efficiently with a **sorted list**.
- Binary search technique searches "value" in **minimum possible** comparisons.
- the given array is a sorted one, otherwise first we have to sort the array elements. Then apply the following conditions to search a "value".
  1. Find the **middle element** of the array (i.e., n/2 is the middle element if the array or the sub-array contains n elements).
  2. **Compare** the middle element with the value to be searched, then there are following three cases.
     a) If it is a desired element, then search is **successful**.
     b) If it is **less than** desired value, then search only the **first half** of the array, i.e., the elements which come to the **left side** of the middle element.
     c) If it is **greater than** the desired value, then search only the **second half** of the array, i.e., the elements which come to the **right side** of the middle element.
  3. Repeat the same steps until an element is found or exhaust the search area.



Example: Consider an array of seven elements

A

9	10	25	30	40	45	70
0	1	2	3	4	5	6

VAL = 45

Step 1:

BEG MID END

9	10	25	30	40	45	70
0	1	2	3	4	5	6

BEG = 0, END = 6

MID = (BEG + END)/2 = (0+6)/2 = 3

A[MID] = A[3] = 30

if(A[3] == VAL) (F)

A[3] < VAL, second half (right)

Now change BEG = MID + 1

= 3 + 1 = 4

Step 2:-

BEG END

9	10	25	30	40	45	70
0	1	2	3	4	5	6

BEG = 4, END = 6

MID = (4+6)/2 = 5

A[MID] = A[5] = 45

if(A[5] == VAL) (T) ✓

element found

END left → mid - 1

BEG right → mid + 1

Algorithm for Binary search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Binary Search Algorithm.

worst case - $O(\log n)$

best case - $\Omega(1)$

Average case - $\theta(\log n)$

Exercise -1 (Searching)

b)Write C program that use both recursive and non recursive functions to perform binary search for a Key value in a given list.

INTERPOLATION SEARCH

a[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}

Low = 0, High = 10, VAL = 19, a[Low] = 1, a[High] = 21

Middle = Low + (High – Low)×((VAL – a[Low]) /(a[High] – a[Low]))

= 0 +(10 – 0) × ((19 – 1) / (21 – 1))

= 0 + 10 × 0.9 = 9

a[middle] = a[9] = 19 which is equal to value to be searched.

Fibonacci search

- The **Fibonacci search** a**lgorithm** is another variant of binary search based on divide and conquer technique. The binary search as you may have learned earlier that split the array to be searched, exactly in the middle recursively to search for the value. Fibonacci search on the other hand is bit unusual and uses the ***Fibonacci sequence*** to search the value.
- Like binary search, the *Fibonacci search* also work on **sorted array** (non-decreasing order).
- Let the length of given array be **n [0...n-1]** and the element to be searched be **x**.
- Then we use the following steps to find the element with minimum steps:

1. Find the **smallest Fibonacci number greater than or equal to n**. Let this number be **fb(m)** [m'th Fibonacci number]. Let the two Fibonacci numbers preceding it be **fb(m-1)** [(m-1)'th Fibonacci number] and **fb(m-2)** [(m-2)'th Fibonacci number].
2. While the array has elements to be checked:
   -> Compare x with i where i = min((offset+fb(m-2),(n-1)), initial offset = -1
   -> If **x** matches, return index value
   -> Else if the element is greater than x, move the third Fibonacci variable two Fibonacci down, indicating removal of approximately two-third of the unsearched array.

      -> Else if the element is less than x, move the third Fibonacci variable one Fibonacci down. Reset offset to index. Together this results into removal of approximately front one-third of the unsearched array.

      Since there might be a single element remaining for comparison, check if fbMm1 is '1'. If Yes, compare x with that remaining element. If match, return index value.

Example:

- **Consider an array a[11]=**

10	20	30	40	45	50	70	80	85	90	100
0	1	2	3	4	5	6	7	8	9	10

let x = 85

- **0,1,1,2,3,5,8,13,21,34,55,…… (Fibonacci series)**
- **Find the smallest Fibonacci number fb(b) greater than or equal to n => n=11 so fb(m) = 13**
- **i=min((offset+fb(m-2)),(n-1)) = min(-1+5, 10)=4**

fb(m-2)	fb(m-1)	fb(m)	offset	i	a[i]	Action
5	8	13	-1	4	45	45<85, fib one down, offset to i
3	5	8	4	7	80	80<85, fib one down, offset to i
2	3	5	7	9	90	90>85, move fib two steps down.
1	1	2	7	8	85	85==85. stop

Algorithm for Fibonacci search

Fibonacci Search (int arr [], int x, int n)

Step 1: [Initialize] SET fbM2 = 0, fbM1 = 1,
fbM = fbM2 + fbM1, offset = -1;
Step 2: Repeat Step 3 while fbM < n
Step 3: SET fbM2 = fbM1;
fbM1 = fbM;
fbM = fbM2 + fbM1;
[END OF LOOP]
Step 4: Repeat Steps 5 and 6 while fbM > 1
Step 5: SET i = min (offset+fbM2,n-1)
Step 6: IF arr[i] < x
SET fbM = fbM1
fbM1 = fbM2
fbM2 = fbM-FbM1
offset = i
ELSE IF arr[i] > x
SET fbM = fbM2
fbM1 = fbM1-fbM2
fbM2 = fbM-fbM1
ELSE return i
Go To Step 9
[END OF IF]
[END OF LOOP]
Step 7: IF fbM1 && arr[offset+1] = x
return offset+1;
Go To Step 9
[END OF LOOP]
Step 8: return -1
Step 9: Exit

Complexity of Fibonacci Search Algorithm.

worst case: $O(\log n)$

best case: $\Omega(1)$

average case: $\Theta(\log n)$

INTRODUCTION TO SORTING

- Sorting means arranging the elements of an array so that they are placed in some relevant **order** which may be either **ascending** or **descending**.
- A **sorting algorithm** is defined as an algorithm that puts the elements of a list in a certain order, which can be either **numerical order**, **lexicographical order.**
- Efficient sorting algorithms are widely used to optimize the use of other algorithms like **search** and **merge algorithms** which require sorted lists to work correctly.

- There are two types of sorting:
  1. **Internal sorting** which deals with sorting the data stored in the computer's memory
  2. **External sorting** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory

BUBBLE SORT

- **Bubble sort** is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order).

- In **bubble sorting**, consecutive adjacent pairs of elements in the array are compared with each other.
- If the element at the **lower index** is **greater than** the element at the **higher index**, the two elements are **interchanged** so that the element is placed before the bigger one.
- This process will continue till the list of **unsorted elements exhausts**.

- This procedure of sorting is called bubble sorting because elements '**bubble**' to the top of the list.
- Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).
- If the elements are to be sorted in **descending order**, then in first pass the smallest element is moved to the highest index of the array.

Example:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.
(b) Compare 52 and 29. Since 52 > 29, swapping is done.
30, **29, 52**, 87, 63, 27, 19, 54
(c) Compare 52 and 87. Since 52 < 87, no swapping is done.
(d) Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, **63, 87**, 27, 19, 54
(e) Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, **27, 87**, 19, 54
(f) Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, **19, 87**, 54
(g) Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

(a) Compare 30 and 29. Since 30 > 29, swapping is done.
29, 30, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52, **27, 63**, 19, 54, 87
(e) Compare 63 and 19. Since 63 > 19, swapping is done.

```
    29, 30, 52, 27, 19, 63, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
    29, 30, 52, 27, 19, 54, 63, 87
```

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

```
Pass 3:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
    29, 30, 27, 52, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
    29, 30, 27, 19, 52, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.
```

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

```
Pass 4:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
    29, 27, 30, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
    29, 27, 19, 30, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.
```

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

(a) Compare 29 and 27. Since 29 > 27, swapping is done.
27, 29, 19, 30, 52, 54, 63, 87

(b) Compare 29 and 19. Since 29 > 19, swapping is done.
27, **19, 29,** 30, 52, 54, 63, 87

(c) Compare 29 and 30. Since 29 < 30, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

(a) Compare 27 and 19. Since 27 > 19, swapping is done.
19, 27, 29, 30, 52, 54, 63, 87

(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

Algorithm for Bubble Sort

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For I = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

Complexity of Bubble Sort Algorithm.

worst case $O(n^2)$

best case $\Omega(n)$

average case $\Theta(n^2)$

INSERTION SORT

- **Insertion sort** is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time.
- The main idea behind insertion sort is that it inserts each item into its **proper place** in the final list.
- The insertion sort algorithm divides the list into two parts **sorted sub-list** and **unsorted list**.
- In Insertion sort the **unsorted elements** are moved and inserted into the **sorted sub-list** (in the same array) in its appropriate place.
- Initially the sorted part contains only **one element**.
- In each pass, one element from the unsorted list is inserted at its **correct position** in the sorted list.
- As a result, the sorted list **grows** by one element and the unsorted list **shrinks** by one element

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores **sorted values** and another that contains **unsorted values**.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1.
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example:

Consider an unsorted list in an array.

A	12	19	33	26	29	35	22	37
	0	1	2	3	4	5	6	7

Initially the sorted sub-list contains only one element and remaining in unsorted list of same array as shown in below figure.

A	12	19	33	26	29	35	22	37
	0	1	2	3	4	5	6	7

Sorted list unsorted list

Pass 1: Take the first element 19 from unsorted list and place them in correct position in the sorted list. Here 19 is greater than 12 (19 > 12). So no need to exchange the elements place them in same order in sorted list. This is shown in the following figure.

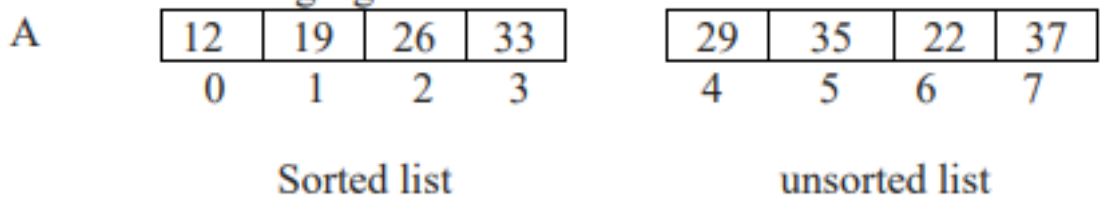
A	12	19	33	26	29	35	22	37
	0	1	2	3	4	5	6	7

Sorted list unsorted list

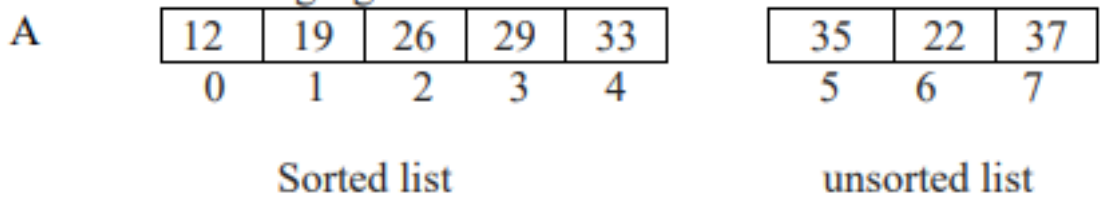
Pass 2: Take the first element 33 from unsorted list and place them in correct position in the sorted list. Here 33 is greater than 19 (33 > 19). So no need to exchange the elements place them in same order in sorted list. This is shown in the following figure.

A	12	19	33	26	29	35	22	37
	0	1	2	3	4	5	6	7

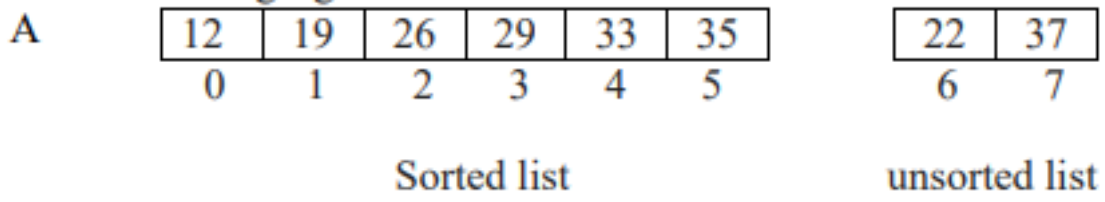
Pass 3: Take the first element 26 from unsorted list and place them in correct position in the sorted list. Here 26 is lesser than 33 (26 < 33) and greater than 19(26 > 19). So shift 33 element at index 3 and store 26 at index 2. This is shown in the following figure.



Pass 4: Take the first element 29 from unsorted list and place them in correct position in the sorted list. Here 29 is lesser than 33 (29 < 33) and greater than 26(29 > 26). So shift 33 element at index 4 and store 29 at index 3. This is shown in the following figure.



Pass 5: Take the first element 35 from unsorted list and place them in correct position in the sorted list. Here 35 is greater than 33 (35 > 33). So no need to exchange the elements place them in same order in sorted list. This is shown in the following figure.



Pass 6: Take the first element 22 from unsorted list and place them in correct position in the sorted list. Here 22 is lesser than 35 (22 < 35), 22 is lesser than 33 (22 < 33), 22 is lesser than 29 (22 < 29), 22 is lesser than 26 (22 < 26) and greater than 19(22 > 19). So shift 35 element at index 6, 33 element at index 5, 29 element at index 4, 26 element at index 3 and store 22 at index 2. This is shown in the following figure.

A

12	19	22	26	29	33	35		37
0	1	2	3	4	5	6		7

Sorted list unsorted list

Pass 7: Take the first element 37 from unsorted list and place them in correct position in the sorted list. Here 37 is greater than 35 (37 > 35). So no need to exchange the elements place them in same order in sorted list. This is shown in the following figure.

A

12	19	22	26	29	33	35	37	---
0	1	2	3	4	5	6	7	

Sorted list **Unsorted list**

Pass 8: There are no elements in unsorted list so the list is completely sorted. The final sorted array is shown in the following figure

A

12	19	22	26	29	33	35	37
0	1	2	3	4	5	6	7

Sorted array of elements.

Algorithm for Insertion Sort

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:      SET TEMP = ARR[K]
Step 3:      SET J = K - 1
Step 4:      Repeat while TEMP <= ARR[J]
                    SET ARR[J + 1] = ARR[J]
                    SET J = J - 1
             [END OF INNER LOOP]
Step 5:      SET ARR[J + 1] = TEMP
         [END OF LOOP]
Step 6: EXIT
```

Complexity of Insertion Sort Algorithm.

a. In worst case:

Consider a list

38	32	28	24	15	10	8	1
0	1	2	3	4	5	6	7

The list comparison starts from 2nd element and moves to 1st element and for next iteration it starts from 3rd element and moves to first element and so on.

From algorithm: for i=1 number of comparisons is 1 and number of shifting's is 1

i.e. For i=1 → 1 + 1 =2
i=2 → 2 + 2 =4
i=3 → 3 + 3 =6
i=4 → 4 + 4 =8
---.
---.
---.
i=n-1 → (n-1) + (n-1) = 2(n-1)

Therefore total is 2+4+6+8+ --- - - - - -- - - - +2(n-1)

= 2(1) +2(2) +2(3) +2(4) + - - - - - - - - - - +2(n-1)

=2(1+2+3+4+ - - - - - - - - - - - -+ n-1)

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

$O(n^2)$

b. In Best case:

Consider a list

1	8	10	15	24	28	32	38
0	1	2	3	4	5	6	7

In this list every time we make one comparison and no shifting of elements because the element is always greater than the previous one.

From algorithm: for i=1 number of comparisons is 1 and number of shifting's is 0

i.e. For i=1→ 1 + 0 =1
i=2 →1 + 0 =1
i=3→ 1 + 0 =1
i=4 →1 + 0 =1
---.
---.
---.
i=n-1 → 1 + 0 =1

Therefore the algorithm make totally **n** comparisons

In best case this algorithm takes $\Omega(n)$ **times.**

Time complexity:

Best case: ~~$\Theta(n)$~~ $\Omega(n)$
Average case: ~~$\Omega(n^2)$~~ $\Theta(n^2)$
Worst case: $O(n^2)$

SELECTION SORT

- Selection sort is generally used for sorting files with very large objects (records) and small keys.
- Consider an array **ARR** with **N** elements. Selection sort works as follows:
- First find the **smallest** value in the array and place it in the **first** position. Then, find the **second smallest** value in the array and place it in the **second** position. Repeat this procedure until the entire array is sorted. Therefore,
  - In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
  - In Pass 2, find the position POS of the smallest value in sub-array of N–1 elements. Swap ARR[POS] with ARR[1]. Now, ARR[0] and ARR[1] is sorted.
  - In Pass N–1, find the position POS of the smaller of the elements ARR[N–2] and ARR[N–1]. Swap ARR[POS] and ARR[N–2] so that ARR[0], ARR[1], ..., ARR[N–1] is sorted.

Example:

	0	1	2	3	4	5	6	7
ARR	39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Algorithm for Selection Sort

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
          to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
       [END OF LOOP]
Step 4: EXIT
```

Complexity of Selection Sort Algorithm.

Worst case = $O(n^2)$

best case = $\Omega(n^2)$

Average case = $\Theta(n^2)$

QUICK SORT

- The quick sort was developed by **C.A.R. Hoare**, this is the **best average** behavior among all sorting methods.
- This algorithm is based on the **divide and conquer** approach that involves successively dividing the problem into smaller problems, until the problems become so small that they can be directly solved.
- The quick sort algorithm works by selecting an element from the list called a **pivot** and then partitioning the list into **two parts** that may or may not be equal.
- The list is partitioned by rearranging the elements in such a way that all the elements towards the left end of the list are smaller than the pivot and all the elements towards the right end of the list are greater than the pivot. The pivot is then placed at its **correct position** between the two sublists.
- This process is repeated for each of the two sub-lists created after partitioning and the process continues until one element is left in each sub-list.

- **Procedure for quick sort**:

1. Select a starting element as **Pivot** value.
2. Divide the list into two parts, such that one part contains all elements less that or equal to the pivot and the other part contains all elements greater than the pivot.
3. Place the pivot at its correct position between the two parts of the list.
4. The preceding process will stop when there is a maximum of one element in each sublist. At that point the list will be completely sorted.

There are many ways to choose pivot:

1. Always choose 1st element
2. Always choose last element as pivot
3. Choose median of array as pivot (can be chosen in linear time).
4. Choose random element as pivot.

Example:

Consider an unsorted list in an array.

arr	28	55	46	38	16	89	83	30
	0	1	2	3	4	5	6	7

Initially in the give list, take arr [0] as the **pivot**, as shown in the following fiqure.

arr	28	55	46	38	16	89	83	30
	0	1	2	3	4	5	6	7

Pivot

Selecting the pivot value

After selecting the pivot value, perform the following steps:

Step 1: Starting from the left end of the list (at index 1) and moving in the left to right direction search for the first element that is greater than the pivot value. Here, **arr[1]** is the first value greater than the pivot.

Step 2: similarly, starting from the right end of the list, and moving in the right to left direction, search for the first element that is smaller than or equal to the pivot value. Here, **arr[4]** is the first value smaller than pivot. The two searched values are depicted in the following figure.

arr	28	55	46	38	16	89	83	30
	0	1	2	3	4	5	6	7

Greater value(i) **Smaller value(j)**

Array depicting two searched values

Step 3: In the preceding figure, the greater value is on the left hand side of the smaller value. This means that the values are not in the correct order.

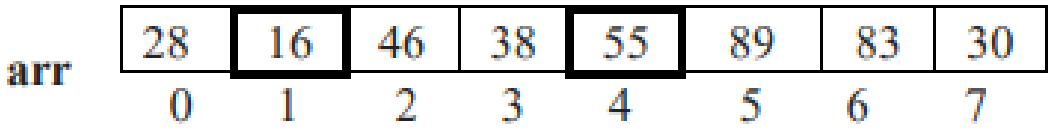
Interchange **arr [1]** and **arr [4]** so that the smaller value is place on the left hand side and the greater value is placed on the right hand side. The resultant array is shown in the following figure.

arr	28	16	46	38	55	89	83	30
	0	1	2	3	4	5	6	7

Array after interchanging values

Step 3: In the preceding figure, the greater value is on the left hand side of the smaller value. This means that the values are not in the correct order.

Interchange **arr [1]** and **arr [4]** so that the smaller value is place on the left hand side and the greater value is placed on the right hand side. The resultant array is shown in the following figure.

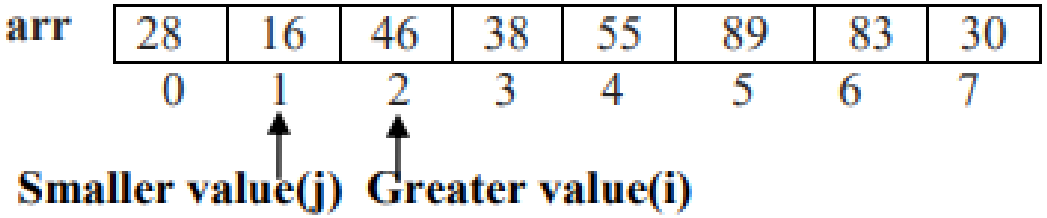


Array after interchanging values

Step 4: continue the search for an element greater than the pivot, starting from **arr[2]** and moving in the left to right direction. Here, **arr[2]** is found to be greater than the pivot.

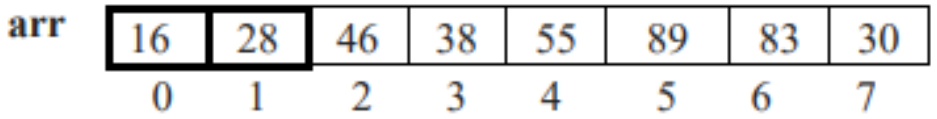
Step 5: Similarly, continue the search for an element smaller than or equal to the pivot starting from **arr[3],** and moving in the right to left direction. Here, **arr[1]** is found to be smaller than the pivot.

The two searched values are depicted in the following figure.



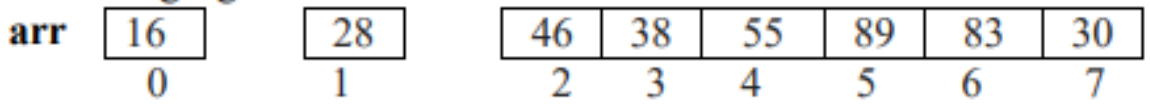
Array depicting two searched values

Step 6: In the preceding figure, the smaller value in on left hand side of greater value. Then replace the pivot value with the smaller value. Interchange **arr[0]** and **arr[1]** this is depicted in the following figure.



Array after interchanging values

Step 7: In the preceding figure the pivot value 28 is placed at its correct position. The elements left to pivot are smaller and the elements right to pivot are greater. Now divide the list into two sub-lists, list1 and list2. List1 contains values lessthan the pivot and list2 contains values greater than the pivot. This is shown in the following figure.

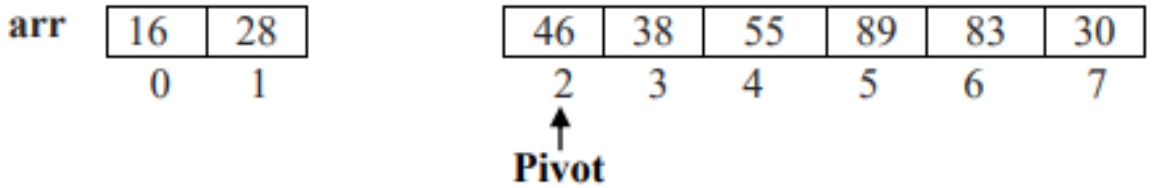


List 1 **list 2**

The two sub-lists

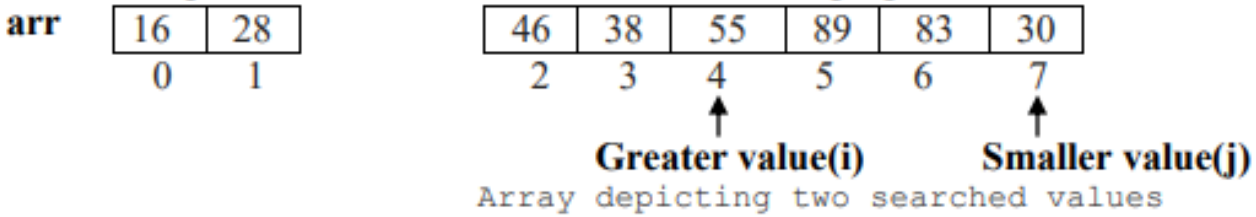
Step 8: In the preceding figure two sub-lists are created this two sub-lists are to be sorted. List 1 contains only one element and we no need to sort the list 1.

Step 9: In List 2 the **arr[2]** that is 46 is considered as pivot. This is shown in the following figure.



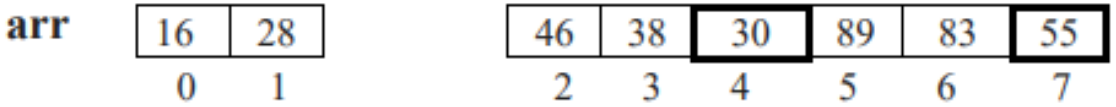
The Pivot value for list 2

Step 10: For list 2 start the same procedure of sorting from step 1 to step 7. Start from left side of the list2 found **arr[4]** is greater than the pivot value. Similarly start from right side of the list2 found **arr[7]** is smaller than the pivot value. this is shown in the following figure.



Array depicting two searched values

Step 11: In the preceding figure, the greater value is on the left hand side of the smaller value. This means that the values are not in the correct order. Interchange **arr [4]** and **arr [7]** so that the smaller value is place on the left hand side and the greater value is placed on the right hand side. The resultant array is shown in the following figure.

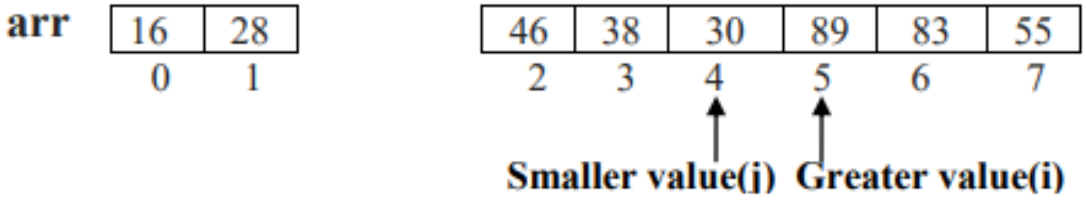


The interchanged values

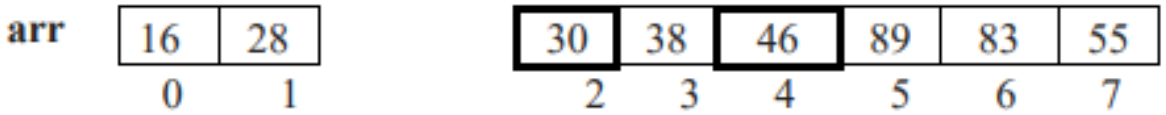
Step 12: continue the search for an element greater than the pivot, starting from **arr[5]**, and moving in the left to right direction. Here **arr[5]** found to be greater than the pivot.

Step 13: Similarly, continue the search for an element smaller than the pivot, starting from **arr[6]**, and moving in the right to left direction. Here **arr[4]** found to be smaller than the pivot.

The two searched values are depicted in the following figure.

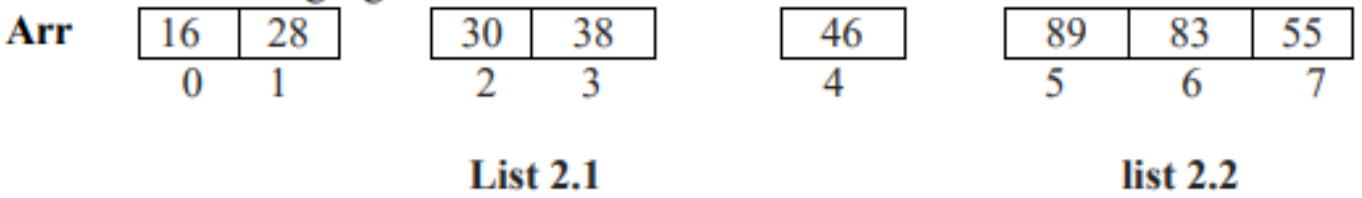


Step 14 : In the preceding figure, the smaller value in on left hand side of greater value. Then replace the pivot value with the smaller value. Interchange **arr[2]** and **arr[4]** this is depicted in the following figure.



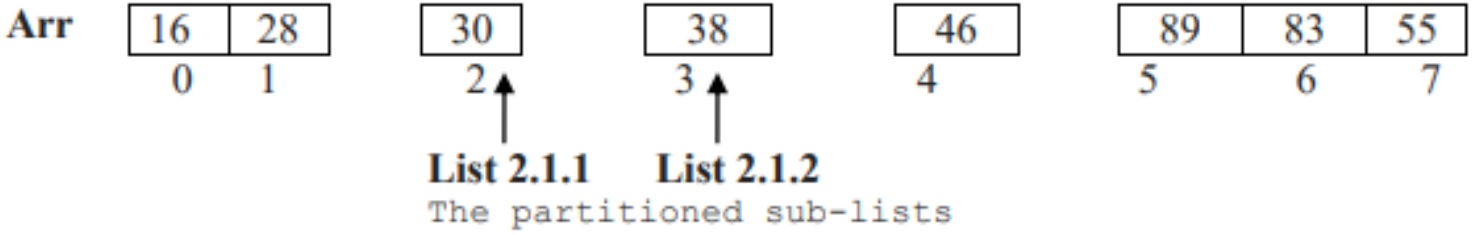
Array after interchanging values

Step 15: In the preceding figure the pivot value 46 is placed at its correct position. The elements left to pivot are smaller and the elements right to pivot are greater. Now divide the list into two sub-lists, list2.1 and list2.2. List 2.1 contains values lessthan the pivot and list2.2 contains values greater than the pivot. This is shown in the following figure.



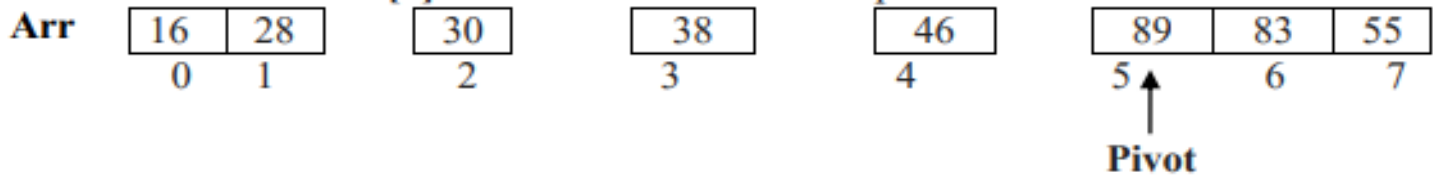
The two sub-lists

Step 16: In the preceding figure two sub-lists are created this two sub-lists are to be sorted. List 2.1 contains two elements, take **arr[2]** that is 30 as pivot and the elements right to it is greater than the pivot and left to is lesser than pivot. Here in list 2.1 contains only one element right to it that is 38 greater than pivot so we no need to swap them and partition the list into two parts and after partitioning we aging sublists with single elements in the list we no need to sort them. This is shown in the following figure.

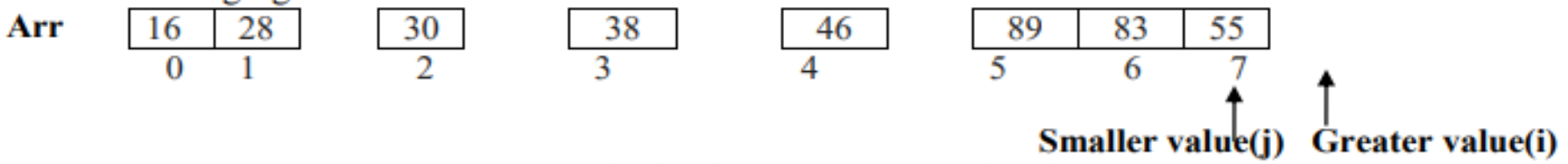


The partitioned sub-lists

Step 17: In List 2.2 the **arr[5]** that is 89 is considered as pivot. This is shown in the following figure.

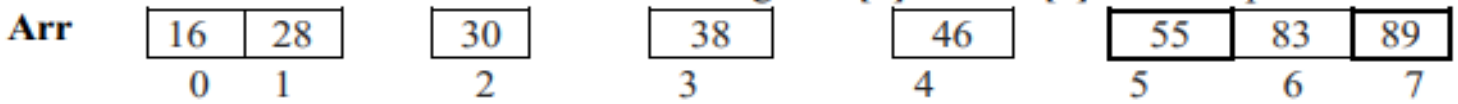


Step 18: For list 2.2 start the same procedure of sorting from step 1 to step 7. Start from left side of the list2.2 found no element is greater than the pivot value and the index is crossed the smaller value . Similarly start from right side of the list2 found **arr[7]** is smaller than the pivot value. this is shown in the following figure.



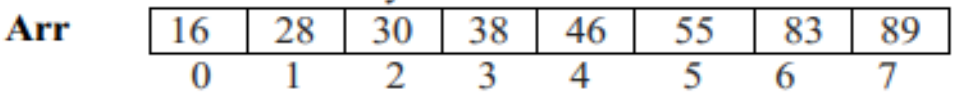
Array depicting two searched values

Step 19: In the preceding figure, the smaller value in on left hand side of greater value. Then replace the pivot value with the smaller value. Interchange **arr[5]** and **arr[7]** this is depicted in the following figure.



Array after interchanging values

Step 20: In the preceding figure the pivot value 89 is placed at its correct position. The elements left to pivot are smaller and the elements right to pivot are greater. In the above figure there are no elements right to pivot so the list is divided into one part as left as list 2.2.1 and it contains two elements from this take 55 as pivot and find the greatest value and smallest value. here 55 smallest value and83 is the greatest value. here pivot is in its correct position and again divide the list and after dividing we found that the list contains only one element and it is already sorted and the final sorted list is shown in the following figure.



Algorithm for Quick Sort

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

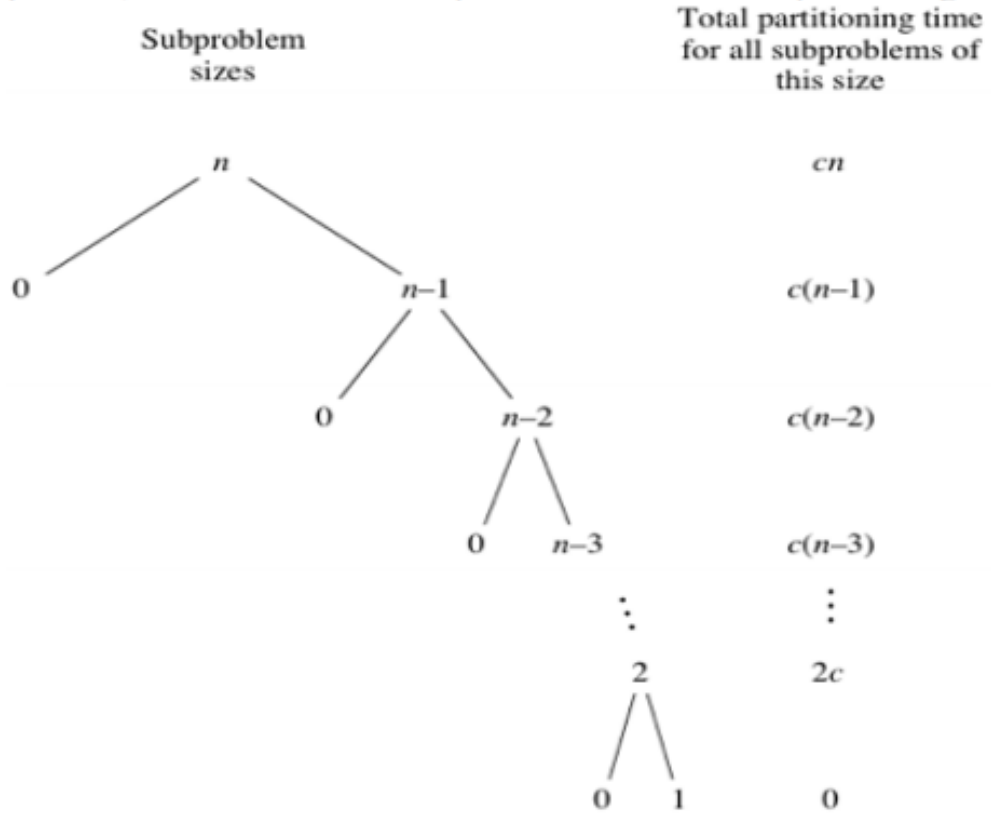
```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
                CALL PARTITION (ARR, BEG, END, LOC)
                CALL QUICKSORT(ARR, BEG, LOC - 1)
                CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Complexity of Quick Sort Algorithm.

a. In worst case:

Consider a list **2 4 8 10 16 18 27**

In the above list always partitioning is done at the beginning of the list. Then the list has the most unbalanced partitions possible, Here's a tree of the subproblem sizes with their partitioning times:



When we total up the partitioning times for each level, we get

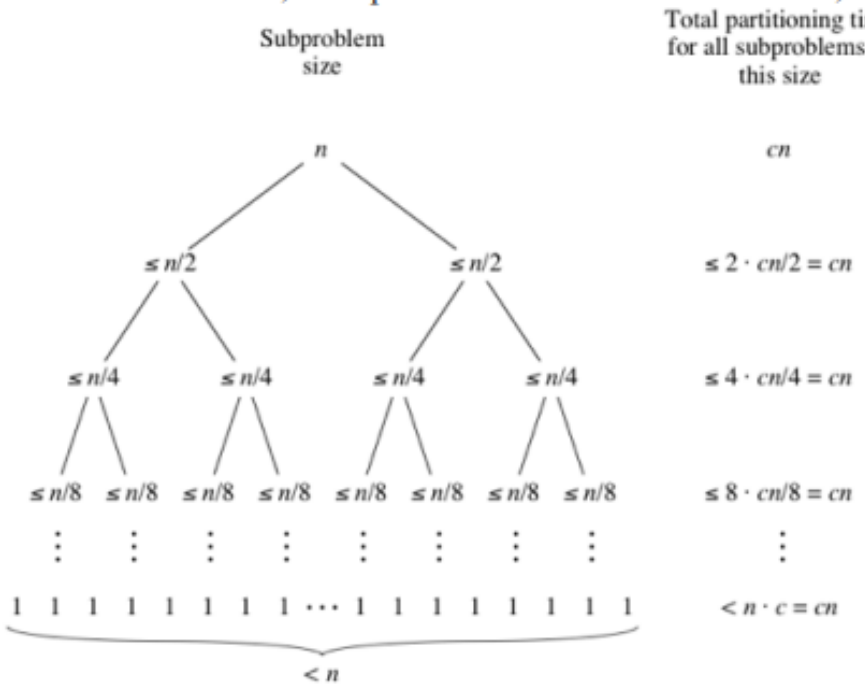
cn+c(n−1)+c(n−2)+⋯+2c take constant C as common

Then **c(n+(n−1)+(n−2)+⋯+2)**. this is in arithmetic series then we get

c((n+1)(n/2)−1) We have some low-order terms and constant coefficients, but when we use big-Θ notation, we ignore them. In big-Θ notation, quicksort's worst-case running time is **O(n2)**

b. In best case:

Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n−1)/2 elements. The latter case occurs if the subarray has an even number n of elements and one partition has n/2 elements with the other having n/2−1. In either of these cases, each partition has at most n/2 elements, and the tree of sub problem sizes looks a lot like tree.



Here the height of the tree is always divided by 2. To get value 1 the tree is divided into 2^k **times**

That is $n/2^k = 1$ therefore for partitioning k levels it takes time is $k = \log_2 n$

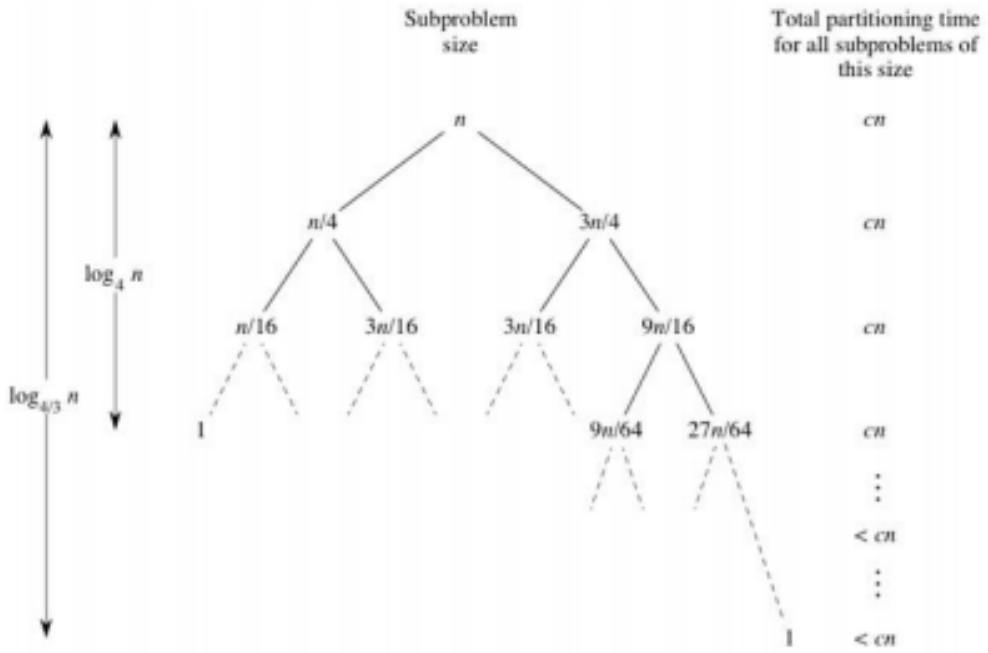
At each level it takes **n times** for partition the list

Therefore total time taken is $\Theta(n \log_2 n)$

c. In Average case:

Showing that the average-case running time is also $\Omega(n\log_2 n)$ let's imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-to-1 split. That is, imagine that each time we partition, one side gets 3n/4 elements and the other side gets n/4. Then the tree of subproblem sizes and partitioning times are shown below figure.

even if we got the worst-case split half the time and a split that's 3-to-1 or better half the time, the running time would be about twice the running time of getting a 3-to-1 split every time. Again, that's just a constant factor, and it gets absorbed into the big-O notation, and so in this case, where we alternate between worst-case and 3-to-1 splits, the running time is $\Omega(n\log_2 n)$



Therefore time complexities are:

Best case: $\Theta(n\log_2 n)$
Average case: $\Omega(n\log_2 n)$
Worst case: $O(n2)$