UNIT-II

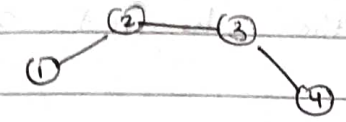
Camlin | Page
Date / /

DISJOINT SETS

Disjoint Sets:

It is also known as union find set. These set donot have any element in common. Conside 2 sets. S_1 and S_2. $S_1 \cap S_2 = \phi$.

Consider 2 seperate set:



$S_1 = \{1, 2, 3, 4\}$

$S_2 = \{5, 6, 7, 8\}$

Main two set operations are

1) Find operation - It takes one parameter and find out the member present in which set.

ex:- Find(4)
$= S_1$
Find(7)
$= S_2$

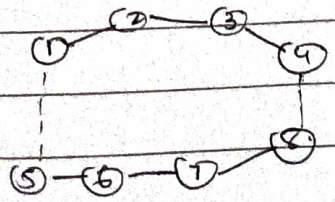
2) Union Operation - In the above example, let us add an edge e(1,5)

Find an union operation mainly useful for detecting cycle in an undirected graph. For the new edge e(1,5) find(1) find(5)

$= S_1$ $= S_2$

Since both the vertices belongs to two diff sets we perform union operation on these two sets. $S_1 \cup S_2$

Let us consider new set as $S_3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Now if we add an edge e(4,8)



Find(4) Find(8)
$= S_3$ $= S_3$

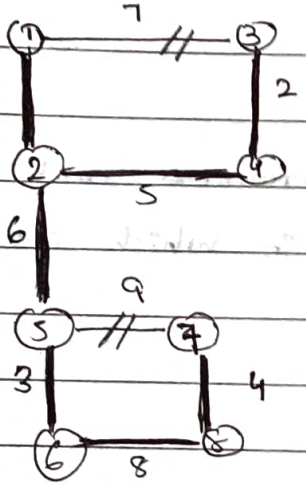
Since both vertices belong to same set the cycle exist upon addition of this edge.

In this way, we can detect occurence of cycle using find and union operation

Application:

Mainly used in Kruskal's algorithm to find cycle in an undirected graph.

Ex:- detection of cycle in given graph using find and union operation.



We start process by considering with min weight e(1,2)

Initially we consider a universal set $U = \{1,2,3,4,5,6,7,8\}$

For the e(1,2) we are creating a set $S_1 = \{1,2\}$

e(3,4)

For this create a set $S_2 = \{3,4\}$

e(5,6)

For this create a set $S_3 = \{5,6\}$

e(7,8)

For this create a set $S_4 = \{7,8\}$

e(2,4) include the edge

Find(2) Find(4)

$= S_1$ $= S_2$

$S_1 \cup S_2$

Since both vertices are in diff sets perform $S_1 \cup S_2$:

$S_5 = \{1, 2, 3, 4\}$

e(2, 5)

Conside e(2, 5)

Find(2) = S_5 Find(5) = S_3

$S_5 \cup S_3$

$S_6 = \{1, 2, 3, 4, 5\}_6$

e(1, 3)

Find(1) = S_6 Find(3) = S_6

Since, those are in same set, the cycle detected & this edge will not be included in the output.

e(6, 8)

Find(6) = S_6 Find(8) = S_4

$S_6 \cup S_4$

$S_7 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

e(5, 7)

Find(5) = S_7 Find(7) = S_7

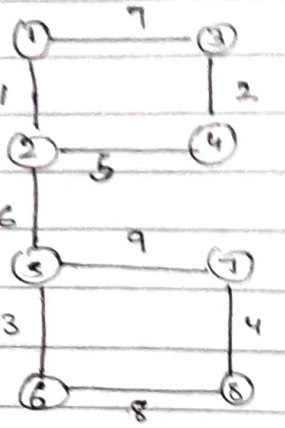
Cycle detected

Representation Of Set :-

Mainly two ways

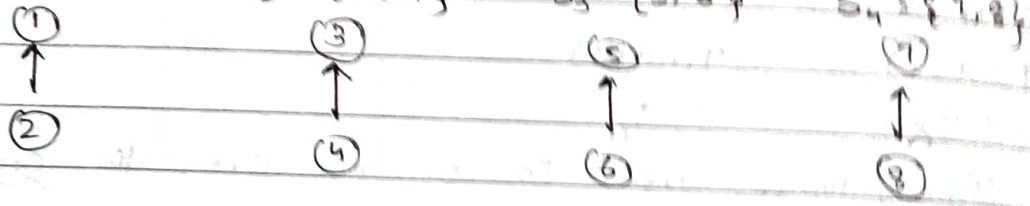
i) Graphical representation

ii) Array representation



Graphical representation:

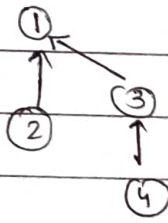
$S_1 = \{1, 2\}$ $S_2 = \{3, 4\}$ $S_3 = \{5, 6\}$ $S_4 = \{7, 8\}$



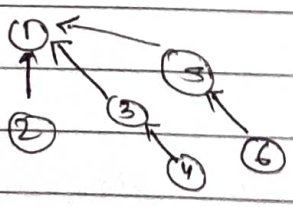
We can take any one element as parent.

Here we took 1 as parent of 2.

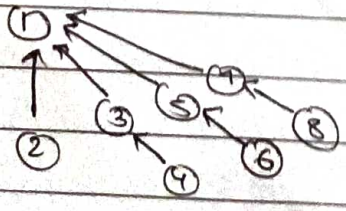
$S_5 = \{1, 2, 3, 4\}$ $S_5 = S_1 \cup S_2$



$S_6 = S_5 \cup S_3 = \{1, 2, 3, 4, 5, 6\}$



$S_7 = S_6 \cup S_4$



Camlin Page
Date / /

⇒ When we perform union if both the set include same no. of nodes, we can retain either the parent of S_1 as new parent (or) parent of S_2 as new parent.

⇒ If the two sets are of diff weights then lesser set with lesser no. of nodes has to combine to the set with higher weight.

Array Representation

For all the vertices a single array called parent ~~array~~ is taken & initialized with a value -1 [To show that initially each vertex is in it's own set]

-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

$S_1 = \{1, 2\}$

① ↑ ②

-2	1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

The index value of 1 is set as -2. (-) represent that node is root; 2 represent no. of nodes in that set with 1 as root.

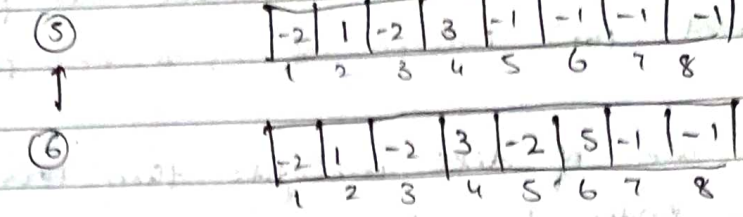
$S_2 = \{3, 4\}$

③ ↑ ④

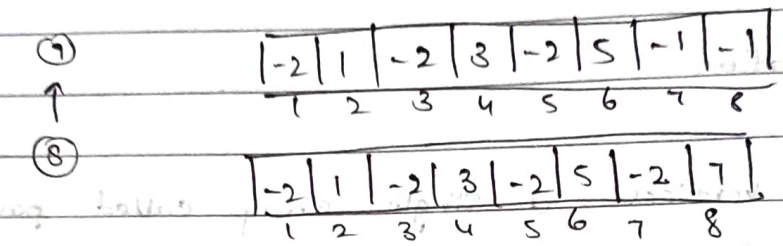
-2	1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

-2	1	-2	3	-1	-1	-1	-1
1	2	3	4	5	6	7	8

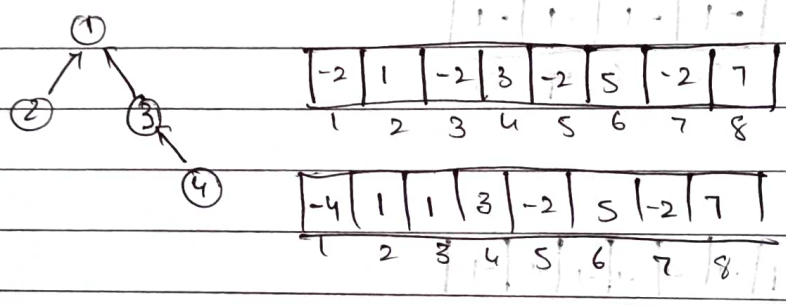
$S_3 = \{5, 6\}$



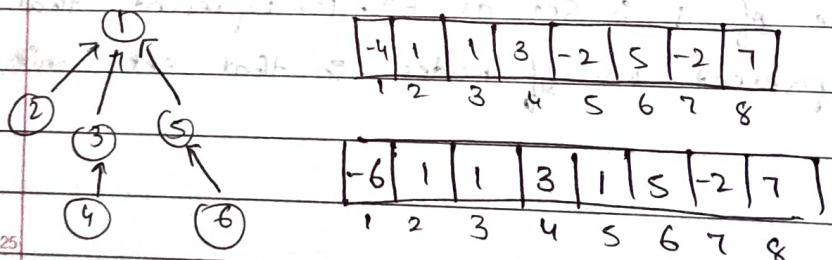
$S_4 = \{7, 8\}$



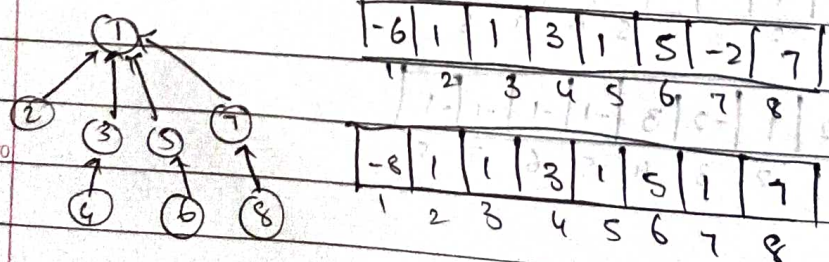
$S_5 = S_1 \cup S_2$



$S_6 = S_5 \cup S_3$



$S_7 = S_6 \cup S_4$



Operations:

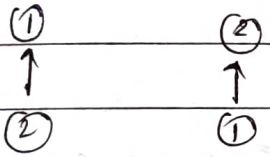
1) Simple Union:

If S_i and S_j are two disjoint sets then $S_i \cup S_j$ = all elements x such that x is either in set S_i (or) in S_j

Algorithm for Simple Union:

```
Algorithm Sunion(i, j)
{
 P[i]=j or P[j]=i
}
```

let us Consider i=1, j=2.



P[j]=i P[i]=j

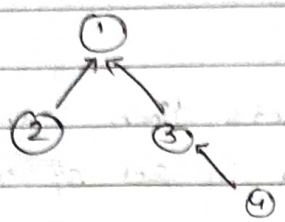
2) Simple find Operation:

for given element i, find operation returns, the set Containing i (root element of that set)

Algorithm for find Operation:

```
Algorithm Find(i)
{
while (P[i]>=0) do
 {
  i=P[i]
 }
return i;
}
```

$S_3 = S_1 \cup S_2$



-4	1	1	3
1	2	3	4.

Find(4) $P[4] = 3 > 0$

i=3 Find(3) $P[3] = 1 > 0$

i=1 Find(1) $P[1] = -4 < 0$

$= -4$

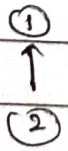
Find(4) = 1

Ex:-

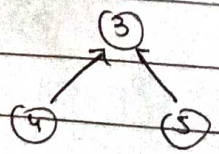
Consider set $S_1 = \{1, 2\}$ $S_2 = \{3, 4, 5\}$. Each set can be represented using the representative element or root element. We can consider any element as representative element for a set. The pointer shows root of that set.

Setno	Ptr
S_1	1
S_2	3

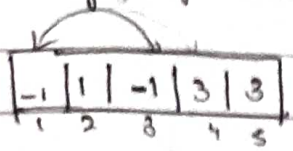
Graphical representation for S_1



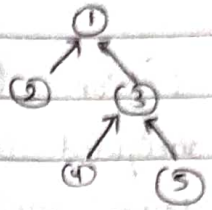
for S_2



The array representation is



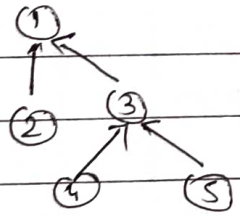
$S_1 \cup S_2$

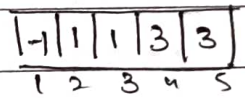


If we perform $S_1 \cup S_2$ in simple union since there is no specific rule either representative element of S_1 can be the root (or) representative element of S_2 can be the root for new set (S_3) ($S_3 = S_1 \cup S_2$). In the above array representation of two sets S_1 and S_2, the weight of the set is not reflected. Only root element is represented not the weight of the set.

Let us Consider we are taking representative element of S_1 as the root of the new set S_3 ($S_1 \cup S_2$).

$S_3 = \{1, 2, 3, 4, 5\}$





The disadvantage of simple find operation is search operation takes (or) find operation takes more time for the elements at deepest level. For example,

In The above set S_3 find(5).

Find (5) $P[5] = 3 > 0$

$i = 3$ Find (3) $P[3] = 1 > 0$

$i = 1$ Find (1) $P[1] = -1 < 0$

Find (5) = 1

To avoid this disadvantage we are using collapsing find.

3) Weighted - Union Operation:

In weighted - union, whenever we perform a union operation, lesser weight set is attached to heavy weight set. In order to, check this condition we add weight information for each representative item for the set.

Algorithm for weighted Union:

```
Algorithm wunion(i,j)
{
 temp = p[i] + p[j]
 if (p[i] > p[j]) then
  {
   p[i] = j
   p[j] = temp
  }
 else
  {
   p[j] = i
   p[i] = temp
  }
}
```

Consider the same set $S_1 = \{1, 2\}$, $S_2 = \{3, 4, 5\}$.

Consider a universal set $\{1, 2, 3, 4, 5\}$. Let us Consider S_1 is form union of 1, 2.

-1	-1	-1	-1	-1
1	2	3	4	5

According to algorithm union(1, 2)

i	j	temp
1	2	-1 -1 = -2
1	3	

P[i] > P[j] -1 > -1 false so else is executed

P[j] = i

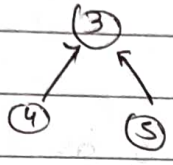
ie., P[2] = 1

P[i] = temp

P[i] = -2

1	2	3	4	5
-2	1	-1	-1	-1

Similarly, using weighted union set S_2 can be represented as



1	2	3	4	5
-2	1	-3	3	3

If we perform $S_1 \cup S_2$, union(1, 3)

i	j	temp
1	3	-2-3 = -5

P[i] > P[j]

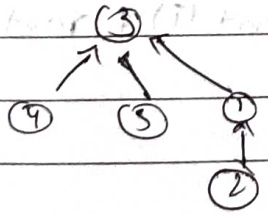
-2 > -3

P[i] = j

P[1] = 3

P[j] = temp

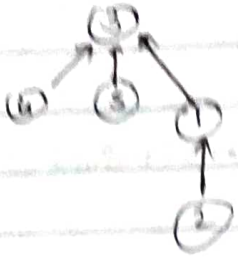
P[3] = -5



1	2	3	4	5
-3	1	-5	3	3

4) Collapsing Find Operation :-

let us Consider a set



3	1	-5	3	3
1	2	3	4	5

According to normal Find operation, if we want to find(2)

$P(2) = 1 > 0$

$P(1) = 3 > 0$

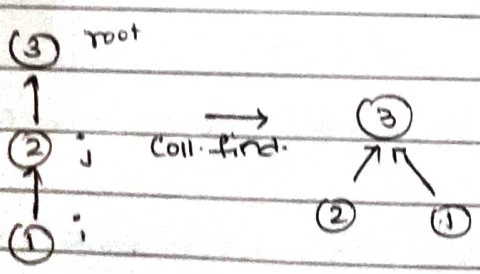
$P(3) = -5 < 0$

So, find operation returns 3 as output.

⇒ Whenever we find nodes at deeper level, we need to perform find operation multiple times To avoid this, nodes at deeper level can be directly connected to representative element of the set

⇒ The procedure of linking a node to a direct parent of the set is known as collapsing find

⇒ Collapsing Find(i) → If j is a node on the path from i to its root (the root represent representative element of set) and if Parent(i) ≠ root(i) then set P[i] = root[i]

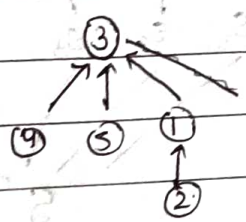


Camlin Page
Date / /

** It reduces the no of operations to find the direct parent (only for the first time it takes more operations from the second operation search onwards it takes less time)

Algorithm for collapsing find:

```
Algorithm cfind(i)
{
  r=i
  while (P[r]>0) do
    r= P[r]
  while (i≠r) do
  {
    S= P[i]
    P[i] = r
    i= s
  }
return s
}
```



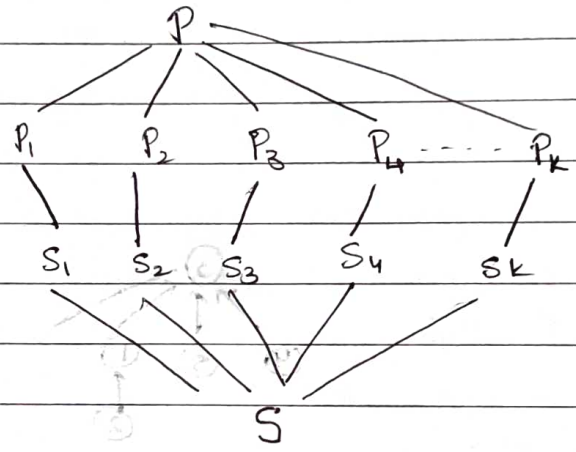
3	1	-5	3	3
1	2	3	4	5

S=P[2]
P[2] = 2
2 = P[2]

i	r	s	
2	2		P[2] >0
	1		r = P[r]
	3		S=P[2]
			=P[1]

Divide & Conquer Strategies:

* It is one of the algorithm design strategy (or technique) where a problem 'P' of large size 'n' is divided into sub problems $P_1, P_2, \ldots P_k$ and find solution for each sub problem $(S_1, S_2 \ldots S_k)$. Now combine all the solutions and find solution for the last problem 'T'



General method:

```
Algorithm DAC(P)
{
  if (Small (P))
    S(P)
  else
  {
    divide probto P1, P2, ... Pk
    Apply DAC(P1), DAC(P2), ...
    Combine((S1), (S2), ... (Sk))
  }
}
```

⇒ It solve the problem recursively.

⇒ Problems solved using divide and conquer strategy are

Camlin Page
Date / /

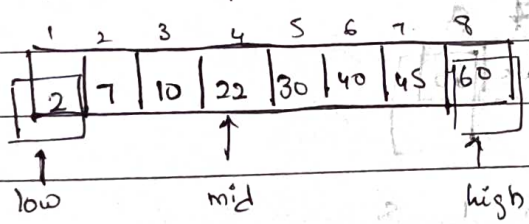
1) Binary Search
2) Merge Sort
3) Quick Sort
4) Strassen's Matrix Multiplication

1 Binary Search:

We perform binary search in a sorted list. In this process we find middle element of list which divide the list into two half.

- If the element to be searched is found in middle index, that index is returned as the result.
- If the element to be searched is greater than the mid index element we perform the search process in the right half
- If the element to be searched is lesser than the mid index element we perform the search process in the left sub array.

This process repeats till the entered array is searched.



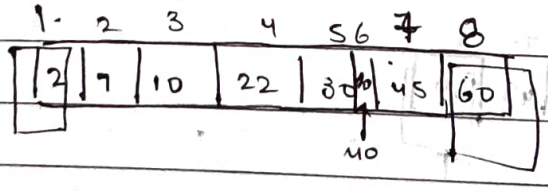
$$mid = \frac{1+8}{2} = 4.$$

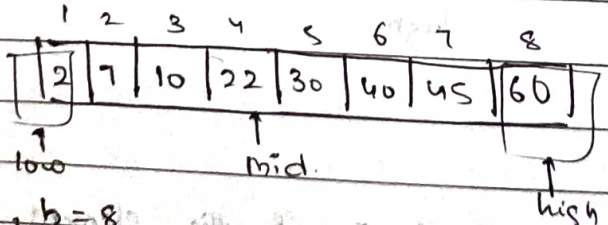
- The best case for binary search is if the element to be searched is present in middle index. In this example, search of 22. Big-Oh (1)
- The worest case for binary search is if the element is not found, element present in first index (or) last index.

⇒ Binary Search Iterative Method:

```
Algorithm IBS(A, n, key)
{
    low = 1, high = n
    while (low <= high)
    {
        mid = (low + high) / 2
        if (key == A[mid]) // if element is found in middle
            return mid;
        else if (key < A[mid]) // To perform search process in left
            high = mid - 1;        sub tree for element less than mid index
        else // To perform search process in right sub array if
            low = mid + 1;         element is greater than mid index value.
    }
    return -1; // if element is not found unsuccessfully
}
```

From the above values, Search value, 10





l=1, h=8

$$mid = \frac{low + high}{2}$$

$$= \frac{1+8}{2} = 4$$

Search = 10

l = 1, h = 8

1 <= 8

mid = $\frac{1+8}{2}$ = 4 (m)

10 ≠ 22

10 < 22

h = m - 1

h = 3

$\frac{1+3}{2}$ = 2 (m)

10 ≠ 7

10 > 7

l = 2 + 1

= 3 (m)

h = 3

m = $\frac{3+3}{2}$ = 3 (m)

3 = 3

10 = A[3] True.

Time Complexity of binary Search using iterative method O(log n)

8/2 = 4 ⟶ $8/2^1 = 4$

4/2 = 2 ⟶ $8/2^2 = 2$

2/2 = 1 ⟹ $8/2^3 = 1$

General formula ⟶ $n/2^k = 1$

$n = 2^k$

$\boxed{k = \log n}$

Binary Search Using Recursive Method

```
Algorithm RBS(low, high, key)
{
 if (low == high)  // for small problems i.e., if there is
 {                          only one element present
  if(A[low] = key)
   return low
 }
 else     // for large problems
  {
  mid = (low + high) / 2

  if (key == A[mid])
  return mid;
  else if (key > a[mid])
  return RBS(mid + 1, high, key);
  else
  return RBS(low, mid - 1, key)
  }
}
```

Search(10) recursively

low = 2 , high = 8

2 ≠ 8

mid = (2+8)/2 = 5

10 ≠ 22

10 > 22 ×

10 < 22

l = 1 h = mid - 1 = 3, 10

mid = (1+3)/2 = 2

10 ≠ 7

10 > 7

h = 3, l = mid + 1 = 2 + 1 = 2, key = 10

10 == 10

True

return 3

Time Complexity of Binary Search recursive method:

Consider Time Complexity of algo as $T(n)$. For small problems i.e., $n=1$ it takes unit time $T(n)=1$.

For large problem, we are dividing list into two half the complexity of sub array is $T(n/2)$ and for other computations let us consider it as unit time i.e., $T(n) = T(n/2) + 1$ for $n > 1$.

So, recurrence relation for Binary Search is

$$T(n) = \begin{cases} 1 & n=1 \\ T(n/2)+1 & n>1 \end{cases}$$

Complexity is $O(\log n)$

Apply master theorem

$a=1 \quad b=2 \quad k=0 \quad P=0$

$\log_b a = 0$

$O(n^k \log^{P+1} n)$

$O(1 * \log n)$

$O(\log n)$

MERGE SORT

In the iterative process of merge sort we perform 2 way merging

Algorithm of merging

```
Alg merging (A, B, m, n)
{
(i=1, j=1, k=1)
while (i<=m && j<=n)
{
 if (A[i] < B[j])
{
 C[k] = A[i]
  k++;
 i++;
}
else
{
  C[k] = B[j]
   k++; j++
}
for(; i <= m; i++)
{
 C[k] = A[i];
 k++;
for(; j<=n; j++)
{
 C[k] = B[j]
  k++;
}
}
```

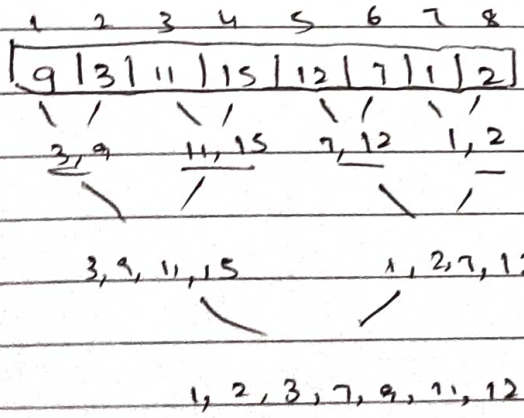
A, B are 2 sorted list m is size of 1st list and n is size of 2nd list. i is an index variable for list A and j for list B and k for new sorted list C.

=> If both the list are of equal length the sorting procedure will be completed within while loop.

=> If 2nd list B contains few more elements than in order to copy the remaining items from B list to resultant list we use 2nd for loop (j loop)

=> If 1st list A contains few more elements than B in order to copy the remaining items of 1st list we use the first for loop (i loop)

Ex for Sorting the given list of elements using merge sort



Initially we take 8 list in the next pass we Consider 4 and then 2 and so on ie.,
no. of passes required is $n/2$ in each phase

Total no. of passes required is $\log \frac{n}{2}$

In each pass we considering n element so the time complexity of merge sort is O(n log n)

Merge Sort Recursive Method

Algorithm mergesort (low, high)

{

if (low < high)

{

mid = $\frac{low + high}{2}$

mergesort (low, mid)

mergesort (mid + 1, high)

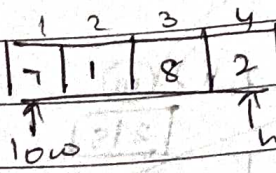
merge (low, mid, high)

}

}

⇒ We consides (l = h) only one element as small input and remaining cases as large input. Whenever we have large size input we use the above recursive algorithm to perform the sorting

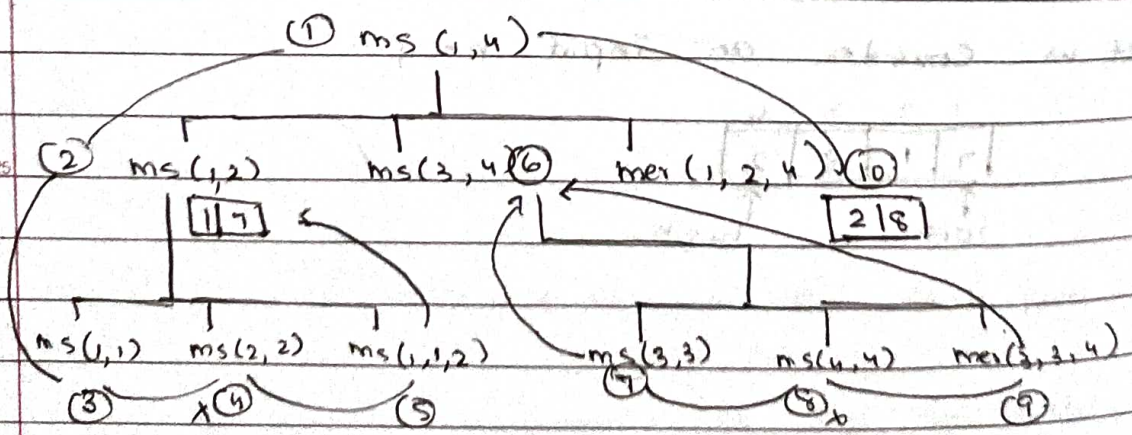
Let us consider an input n = 4



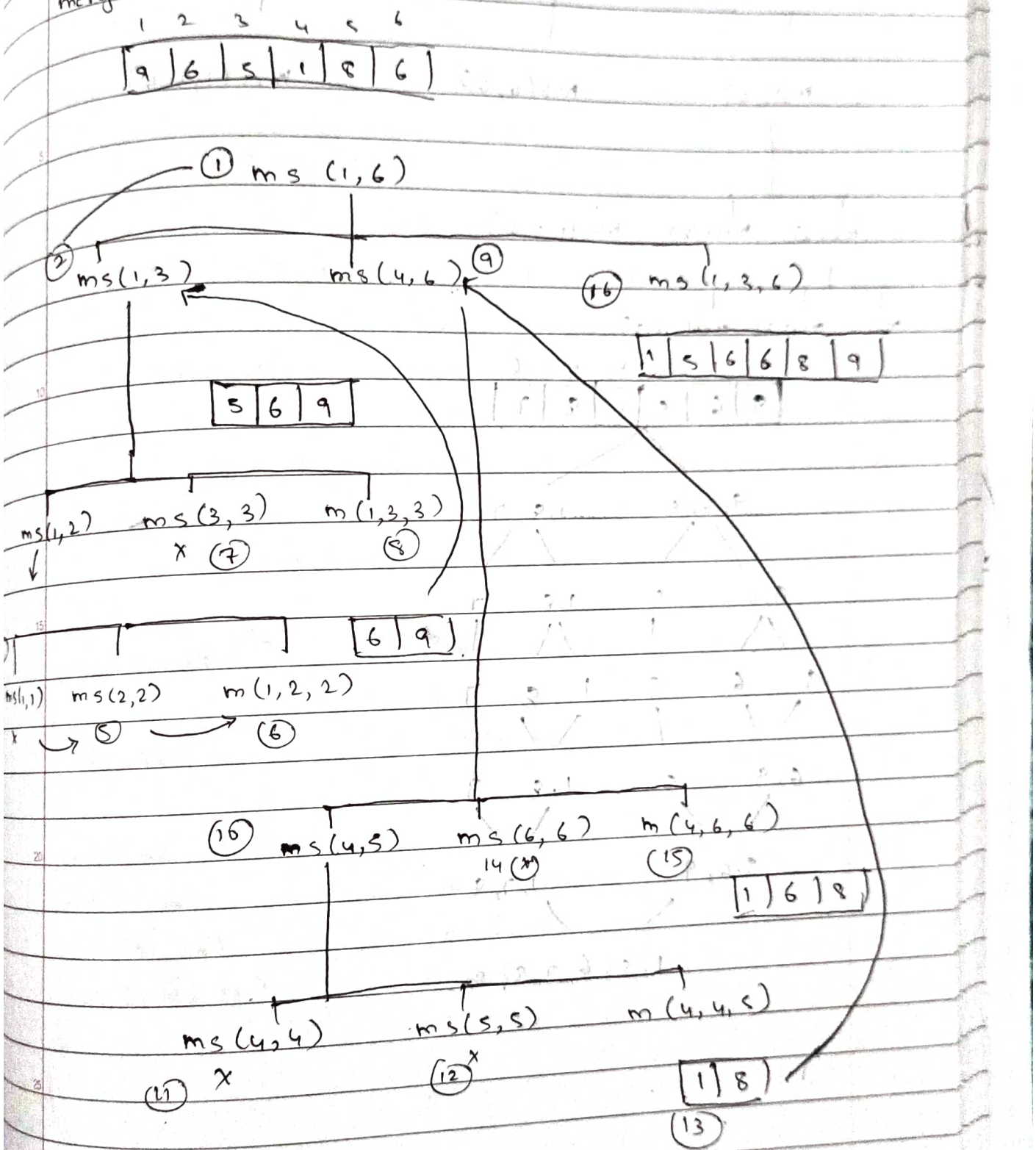
I mergs(1,4)
{
if(1<4)
{
mid = 2
ms(1,2) → II
VI → ms(3,4)
X mer(1,2,4) → [1|2|7|8]
}
}

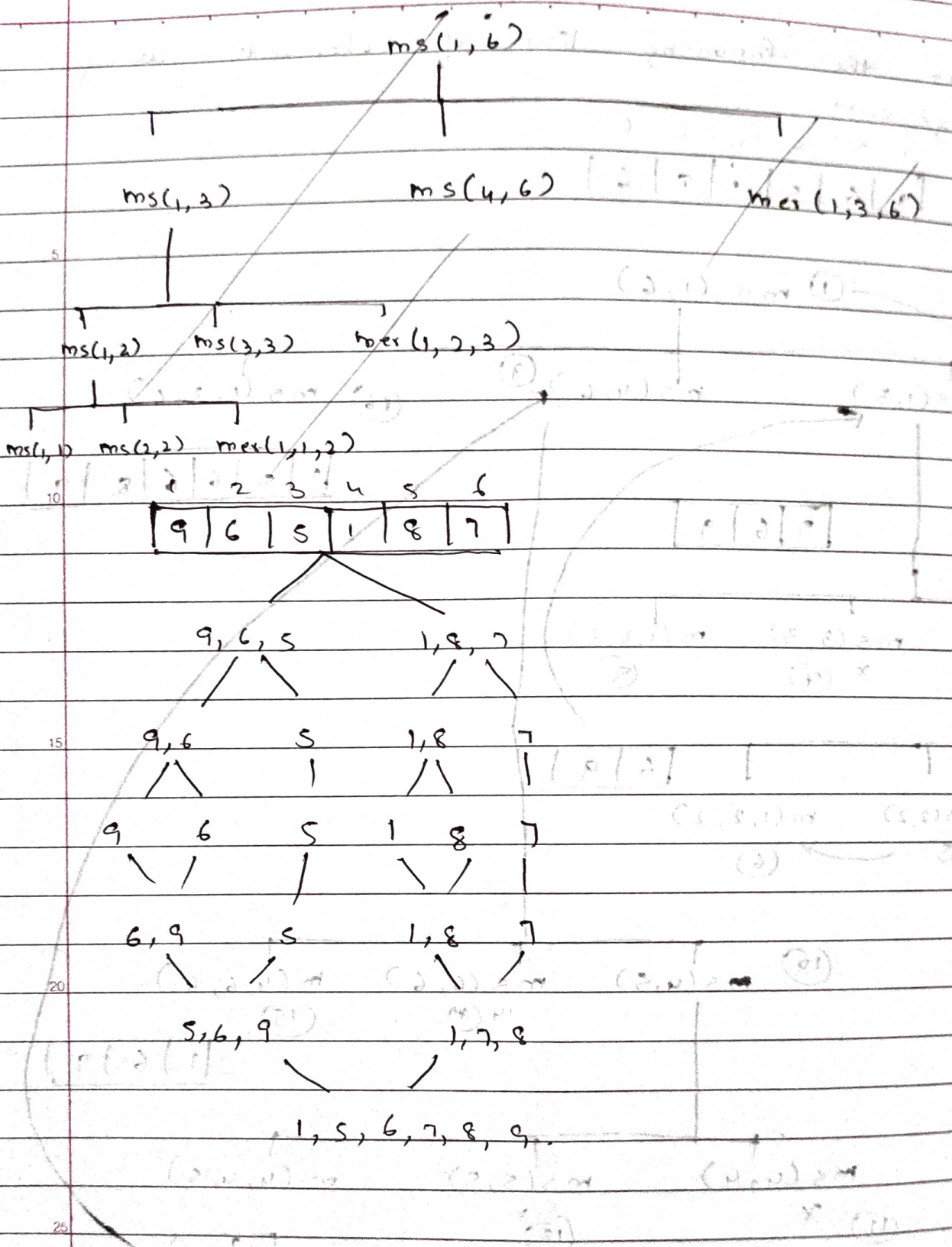
II ms(1,2)
{
if(1<2)
{
mid = 1
ms(1,1) X III
ms(2,2) X IV
X → mer(1,1,2) → [1|7]
}
}

III ms(3,4)
{
if(3<4)
{
mid = 3
ms(3,3) X VIII
ms(4,4) X VIII
IX → mer(3,3,4) → [2|8]
}
}



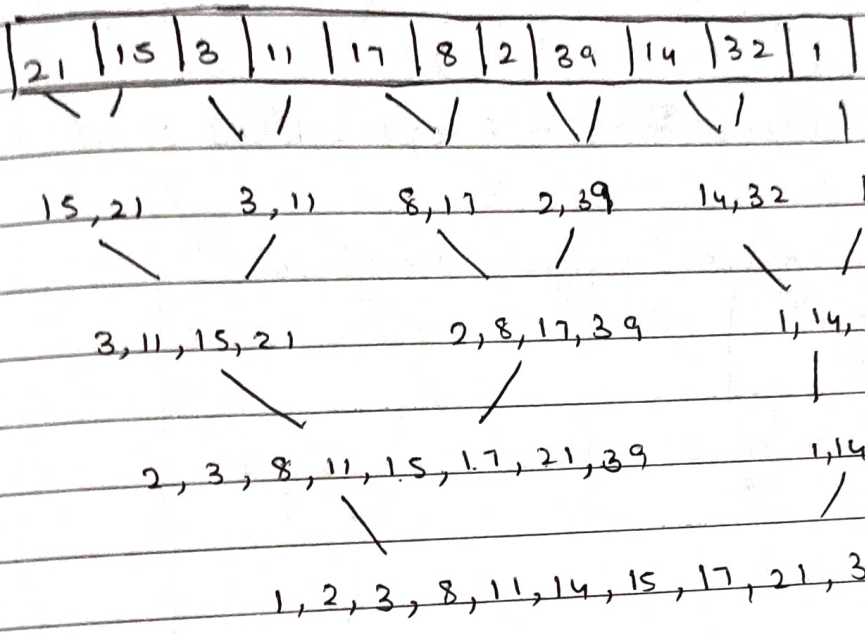
Sort the following list of elements using merge sort




ms(1,6)
ms(1,3)
ms(4,6)
mer(1,3,6)
ms(1,2)
ms(3,3)
mer(1,2,3)
ms(1,1)
ms(2,2)
mer(1,1,2)
1 2 3 4 5 6
9 6 5 1 8 7
9,6,5
1,8,7
9,6
5
1,8
7
9
6
5
1
8
7
6,9
5
1,8
7
5,6,9
1,7,8
1,5,6,7,8,9.

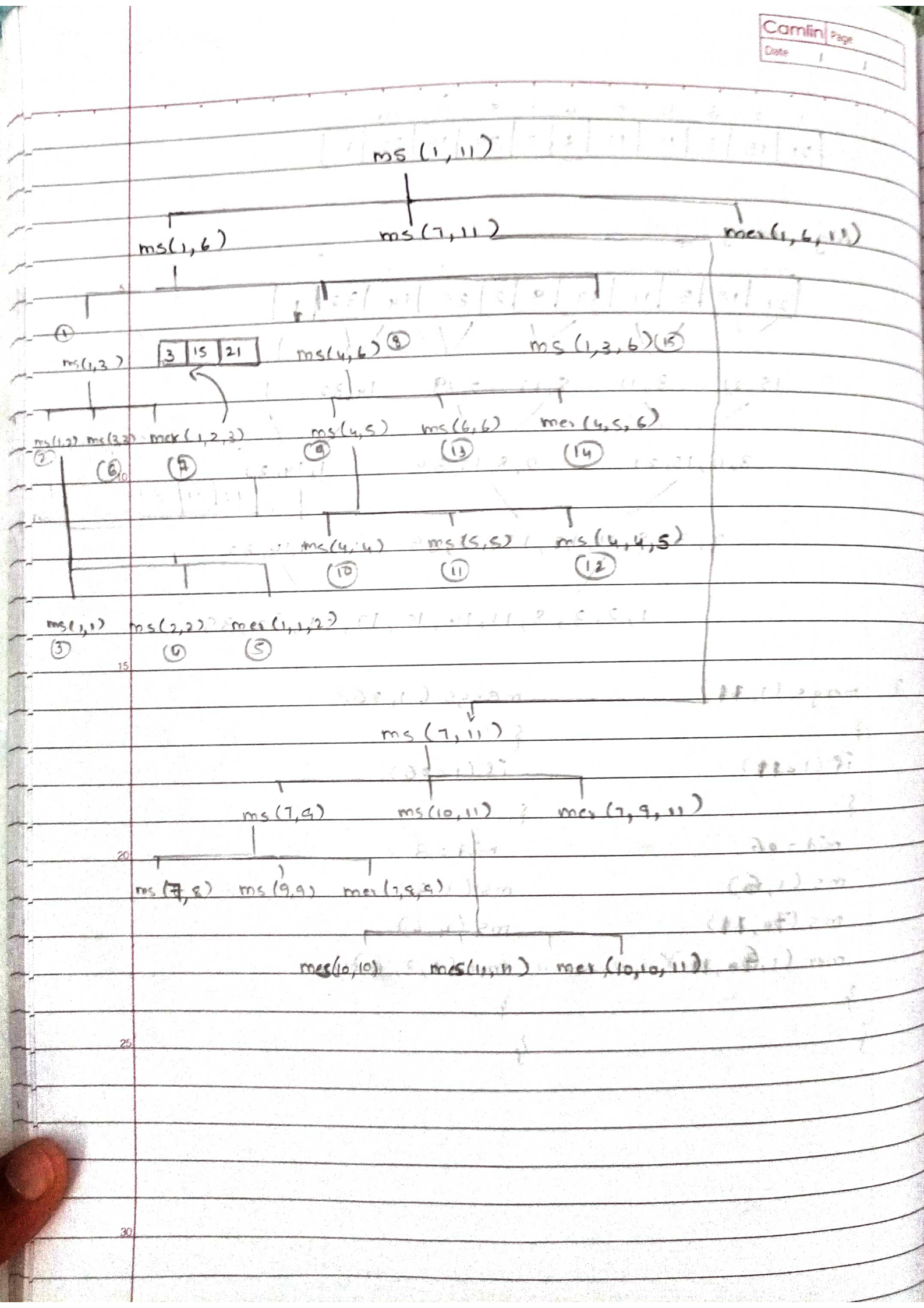
1	2	3	4	5	6	7	8	9	10	11
21	15	3	11	17	8	2	39	14	32	1

Alg ~~merg~~




```
I  mergs (1, ~~11~~)
{
if (1 < ~~11~~)
{
mid = ~~06~~
ms (1, ~~6~~)
ms (~~7~~, ~~11~~)
mer (1, ~~6~~, ~~11~~)
}
}
```

```
mergs (1, ~~06~~)
{
if (1 < ~~06~~)
{
mid = 3
ms (1, 3)
ms (4, 6)
mer (1, 3, 6)
}
}
```


ms (1, 11)
ms(1,6)
ms(7,11)
mer(1,6,11)
ms(1,3)
3 15 21
ms(4,6)
ms(1,3,6)
ms(1,2)
ms(3,3)
mer(1,2,3)
ms(4,5)
ms(6,6)
mer(4,5,6)
ms(4,4)
ms(5,5)
ms(4,4,5)
ms(1,1)
ms(2,2)
mer(1,1,2)
ms (7, 11)
ms(7,9)
ms(10,11)
mer(7,9,11)
ms(7,8)
ms(9,9)
mer(7,8,9)
ms(10,10)
ms(11,11)
mer(10,10,11)

21, 15, 3, 11, 17, 8, 2, 39, 14, 32, 1

1	2	3	4	5	6	7	8	9	10	11
21	15	3	11	17	8	2	39	14	32	1

15, 21 | 11, 3 | 8, 17 | 2, 39 | 14, 32 | 1

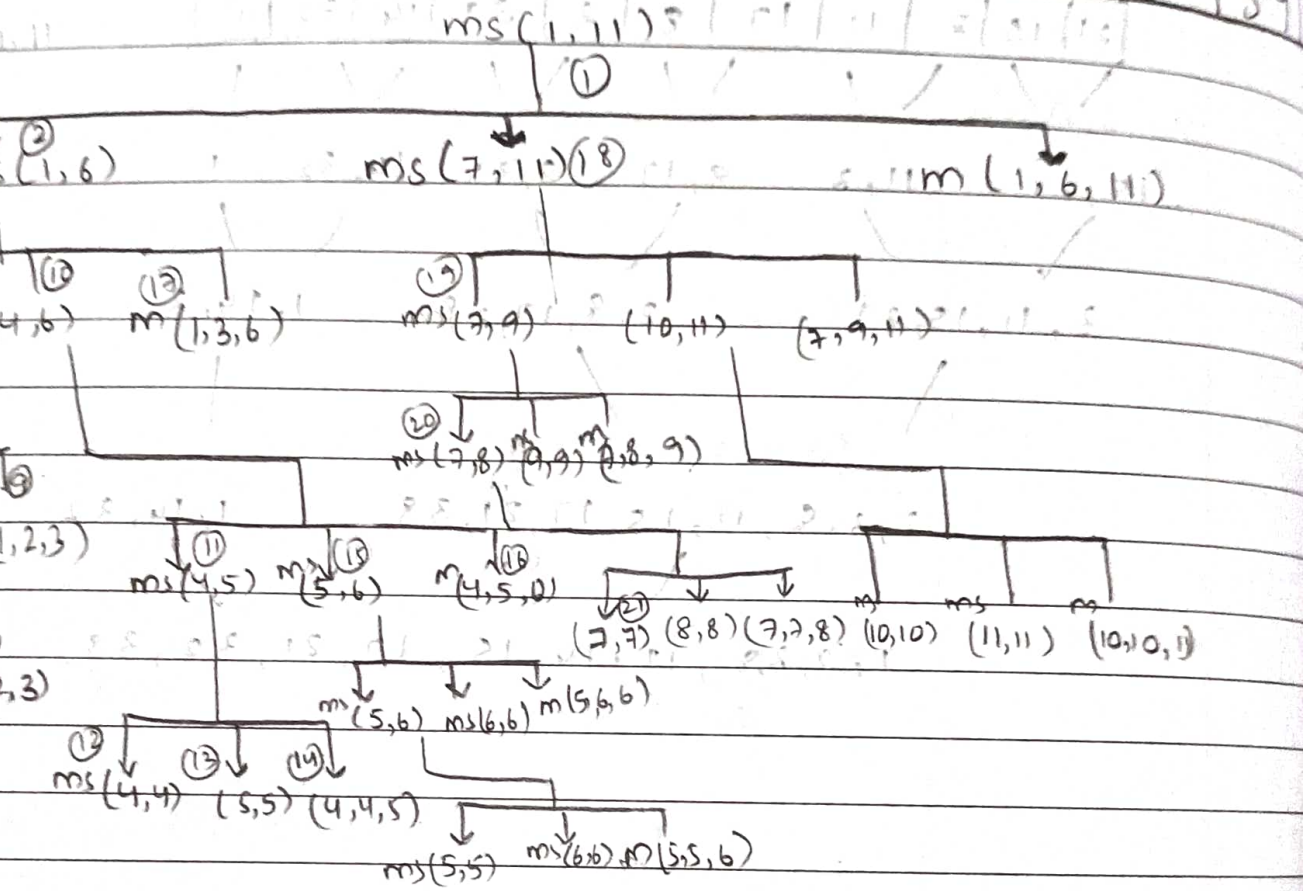
3, 11, 15, 21 | 2, 8, 17, 39 | 1, 14, 32

2, 3, 8, 11, 15, 17, 21, 39 | 1, 14, 32

1, 2, 3, 8, 11, 14, 15, 17, 21, 32, 39.

Tree Method

1	2	3	4	5	6	7	8	9	10	11
1	2	3	8	11	14	15	17	21	32	39



Time Complexity of recursive mergesort:-

For small size of input $n=1$ $T(n) = 1$ $(n=1)$, for large input $n>1$ $T(n) = 2T(n/2) + n$

$T(n) = 1$	$n=1$
$T(n) = 2T(n/2)+n$	$n>1$

$a=2, b=2, f(n)=n, k=1, p=0$

$\log_b^a = k \quad \log_2^2 = 1$

$O(n^k \log^{p+1} n)$

$O(n \log n)$

Strassen's matrix multiplication:-

Consider two 2×2 matrices A, B

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21}$

$c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$

$c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$

$c_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$

For a 2×2 matrix we perform '8' multiplication operation and '4' addition operation.

Algorithm for matrix multiplication is as follows:

```
Algorithm mul(A, B, n)
{
for(i=1; i<=n ; i++)
{
for(j=1; j<=n; j++)
{
    c[i][j]=0
for(k=1; k<=n; k++)
 {
   c[i][j] = c[i][j]+a[i][k]*b[k][j]
 }
}
}
}
```

Time Complexity of normal matrix multiplication is $O(n^3)$.

Consider matrices of larger size like 4×4, 8×8, 16×16 (for ease of division we are considering the rank as power of 2).

ex:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

If we divide the above matrix as shown below

A_{11}		A_{12}			B_{11}		B_{12}	
a_{11}	a_{12}	a_{13}	a_{14}	*	b_{11}	b_{12}	b_{13}	b_{14}
a_{21}	a_{22}	a_{23}	a_{24}		b_{21}	b_{22}	b_{23}	b_{24}
a_{31}	a_{32}	a_{33}	a_{34}		b_{31}	b_{32}	b_{33}	b_{34}
a_{41}	a_{42}	a_{43}	a_{44}		b_{41}	b_{42}	b_{43}	b_{44}
A_{21}		A_{22}			B_{21}		B_{22}	

According to divide and conquer matrix as shown multiplication can be written as

```
Alg mul(A,B,n)
{
if( n<=2 // for smallest input //
{
c11 = a11 x b11 + a12 x b21
c12 = a11 x b12 + a12 x b22
c21 = a21 x b11 + a22 x b21
c22 = a21 x b12 + a22 x b22
}
else // for larger input
{
mid = n/2
mul(A11, B11, n/2) + mul (A12, B21, n/2)
mul (A11, B12, n/2) + mul (A12, B22, n/2)
mul (A21, B11, n/2) + mul (A22, B21, n/2)
mul (A21, B12, n/2) + mul (A22, B22, n/2)
}
}
```

Time Complexity of matrix multiplication using divide and Conquer method:

If the matrix of small dimension it does basic addition and multiplication of elements. So, let us Consider $T(n)$ = Constant time i.e. = 1 for $n \leq 2$

$T(n) = 8T(n/2) + n^2$ for $n \geq 2$

let $T(n)$ be the time Complexity of fn mul. Inside the function we are calling the same fn 8 times by passing $n/2$ elements. So, $8T(n/2)$.

In the same algorithm we are performing 4 matrix addition operation. For matrix addition Complexity is n^2.

$T(n) = 1$ for $n = 2$.

$T(n) = 8T(n/2) + n^2$ $n > 2$.

$a = 8, b = 2, k = 2, p = 0$

$\log_b a = \log_2 8 = 3$.

$\log_b a = k$

$3 > 2$

~~$P > -1$~~

~~$O(a^k \log^{p+1} n)$~~

~~$O(n^2 \log n)$~~.

$= O(n^{\log_b a})$

$= O(n^3)$

Strassen's Matrix Multiplication :

$P = (A_{11} + A_{22}) \times (B_{11} + B_{22})$ [mul the ddalg ele]

$Q = (B_{11} \times (A_{21} + A_{22}))$

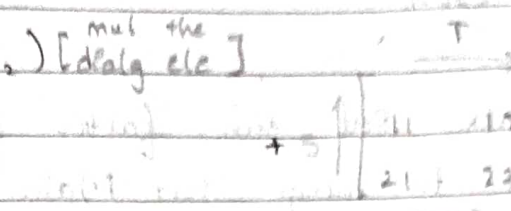
$R = (A_{11} \times (B_{12} - B_{22}))$

$S = (A_{22} \times (B_{21} - B_{11}))$

$T = (B_{22} \times (A_{11} + A_{12}))$

$U = (A_{21} - A_{11}) \times (B_{11} + B_{12})$

$V = (B_{21} + B_{22}) \times (A_{12} - A_{22})$



$C_{11} = P + S - (T + V)$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - (Q + U)$

In the above equations only 7 multiplication operations have been used. So, $T(n) = 1$ for $n = 2$

$T(n) = 7T(n/2) + n^2 \quad n > 2$

$T(n) = 7T(n/2) + n^2$

$a = 7, \; b = 2, \; k = 2, \; p = 0$

$\log_2 7 = 2.8$

$2.8 > 2$

$O(n^{\log_b a}) = O(n^{2.8})$

Quick Sort:

Quick sort run faster for smaller inputs Compared to merge sort. Main procedure for Quick Sort is partioning (The process of placing the pivot element in the corect place. After this process, all the elements towards the left of the pivot element will be less than the pivot (but unsorted) and all the elements towards right side are greater than pivot (but unsorted))

3, 4, 2, 1, [5], 8, 7, 9, 6

Algorithm:-

```
Algorithm QuickSort(l, h)
{
  if (l < h)
  {
    pos = Partitioning(l, h)        pos = position
    QuickSort(l, pos-1);
    QuickSort(pos+1, hi, h);
  }
}
```

Algorithm for Partioning (l, h)

```
Algm part(l, h)
{
Pivot = a[i]
  i = l
  j = h + 1
while (i<=j)
{.
repeat
  {
    i++
  }
until (a[i] >= pivot)
repeat {
    j--
}
until (a[j] <= pivot)
if(i<j)
{
  Swap a[i] and a[j]
  }
}
Swap pivot and a[j]
return j
}
```

Sort the following numbers using quick sort

35, 50, 15, 25, 80, 20, 90, 45

35	50	15	25	80	20	90	45	∞

Pivot

increment i till we find a value which is greater than/equal to 35

35	50	15	25	80	20	90	45	∞
	i							

decrement j until we find a value which is less than or equal to 35

35	50	15	25	80	20	90	45	∞
	i				j			

Check if i<j [2<6] ✓

Swap a[i] & a[j]

35	20	15	25	80	50	90	45	∞

Second iteration

[i<j] [2<6]

increment i till we find a value greater than/equal 35

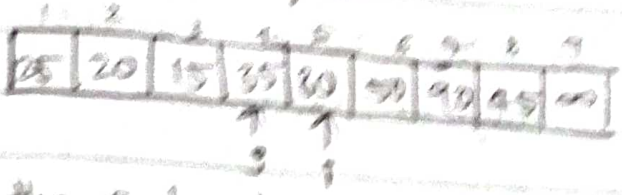
1	2	3	4	5	6	7	8	9
35	20	15	25	80	50	90	45	∞
				i				

decrement j till we find a value less than or equal to 35

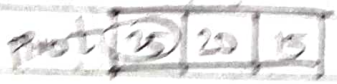
1	2	3	4	5	6	7	8	9
35	20	15	25	80	50	90	45	∞
			j	i				

Check if [i<j] (5<4) false.

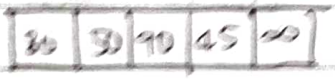
Coming out of while loop
swap a[i] & pivot

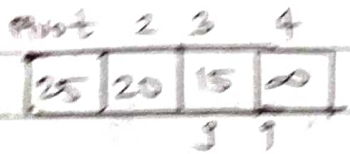


now the pivot elements is in the right place all the values towards left of 35 are less than 35(unsorted) all the values towards right side are greater than 35
Recursively we call quick sort on left side of list.
quicksort(low, pos-1) that is (1,3)
(position is 3rd index[3])

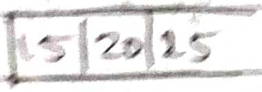


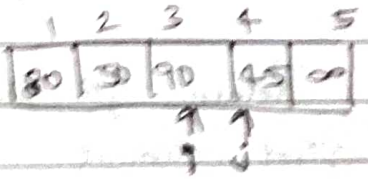
and we recursively perform quicksort of (pos+1, high) that is (5,8)



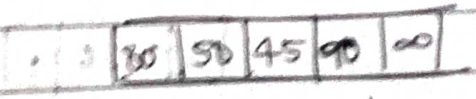


i<j != 4<3 ? ×
swap a[j], pivot

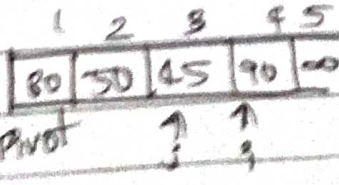




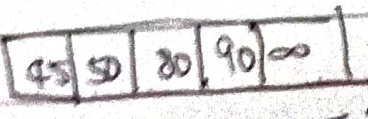
i<j
swap a[i]



second iteration



i<j
swap; a[j] & pivot



THE END

Camlin Page
Date / /

Partioning Algorithm

Step 1:- Initialize pivot as low index element, i as low index and j as high+1 index

Step 2:- Increment the i index till we find a value which is greater than (or) equal to pivot element.

Step 3:- Decrement jth index till we find a value which is less than (or) equal to pivot element

Step 4:- If the ith index is less than jth index, swap a[i] and a[j] & repeat step 2 and 3 till i less than j

Step 5:- If ith index is greater than jth index, swap pivot and a[j]

Step 6:- Return jth index (which is the position of the pivot element

Solve

Sort the following elements using quick sort.

1	2	3	4
5	7	6	1

Quick Sort (1, 4)

{

if (1<4)

{

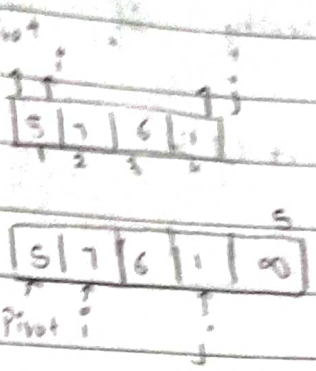
pos = partitioning (1, 4)

Quicksort (1, 1)

Quick (3, 4)

}

}



Quick (1, 1)

{

if (1<1)

x

}

Quick (3, 4)

{

if (3<4)

{

pos = partition (3, 4)

Quick (3, 2)

Quick (4, 4)

}

}

Quick (3, 2)

{

if (3<2)

x

}

Quick (4, 4)

{

if (4<4)

x

}

Time Complexity of Quick Sort:

```
Algo QuickSort (l, h)
{
if (l<h)
{
  pos = partitioning (l, h)
  QuickSort (l, pos-1);
  Quicksort (pos+1, h);
  }
}
```

→ Consider T(n) as time Complexity of Quick Sort We are calling the same function 2 times (for the left half and the right half). Let the time Complexity of these two halves are $T(n/2)$

=> Complexity of partiong algorithm is $O(n)$ [since it Consider entire array].

The recursive formula for Quick Sort

$$T(n) = 1 \quad n=1$$
$$= 2T(n/2) + n \quad n>1$$

2 QuickSort part

$2T(n/2) + n$

$a=2, b=2, k=1, p=0$

$\log_b^a = \log_2^2 = 1$

$\log_b^a = k$

$= O(n^k \log^{p+1} n)$

$= O(n \log n)$

Sort the following index value in the ascending order using quick sort.

2	3	8	5	4	7	6	9	1

QuickSort (1, 9)

{

if (1 < 9)

{

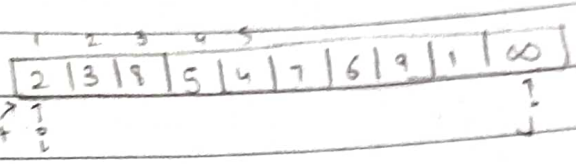
pos = partition (1, 9)

Quick (1, 1)

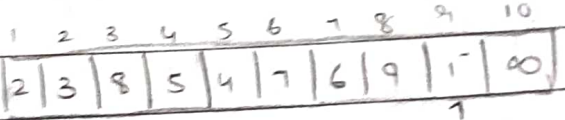
Quick (3, 9)

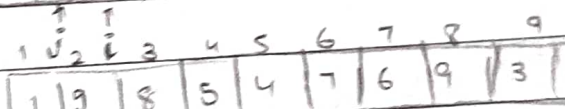
}

}



- increment i till we find a value greater than the pivot
- decrement j till we find a value lesser than pivot





Swap a[j], pivot

Quick (1, 1)

{

if (1 < 1)

{

}

}

Quick (3, 9)

{

if (3 < 9)

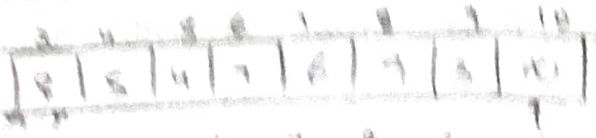
{

pos = part (3, 9)

Quick (3, 7)

Quick (9, 9)

} }

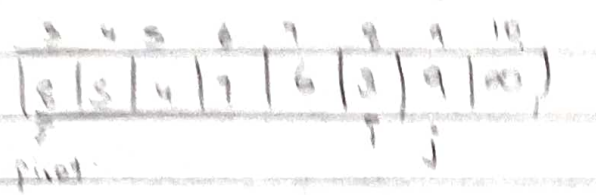


pivot i j

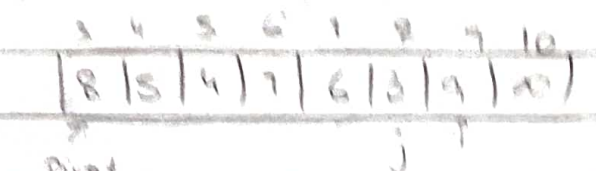
i<j

3<9 true

a[i], a[j] swap



pivot j

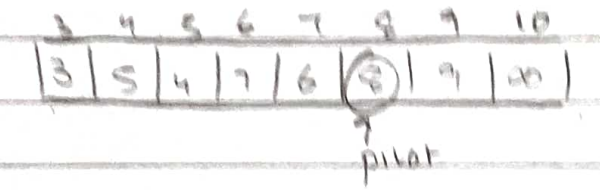


i++ pivot j

i<j

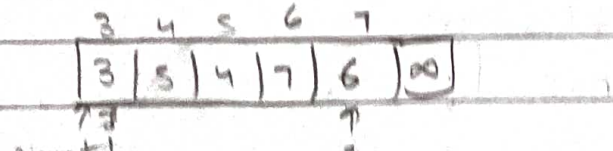
9<9 false

a[i], pivot swap

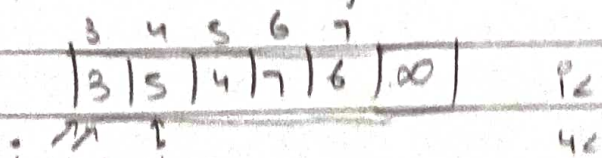


pivot

```
Quick(3,7)
if(3<7)
{
pos = partion(3,7)
Quick(3,2)
Quick(4,7)
}
}
```



pivot+1 j

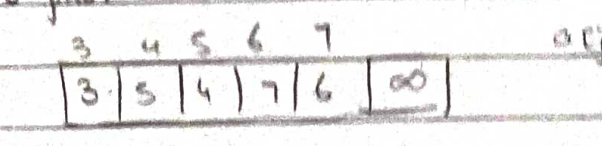


i pivot j

i<j

4<8 false

a[j], pivot



position of pivot is in right place.

Quick(3,2) if(4<2)

x

Quick(4,7)

if(4<7)

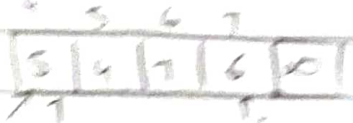
{

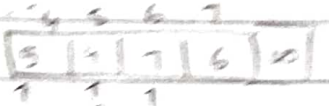
Pos = part(4,7)

Quick(4,x)

Quick(5,7)

}{



i++ 

pivot is in right place

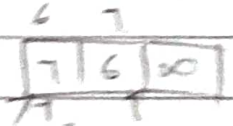
Quick(4,4) Quick(6,7)

if(4<4) if(6<7)

x pos = part(6,7)

} Quick(6,5)

Quick(7,7)



Quick(6,5)

if(6<5)

x

}

Quick(7,7)

if(7<7)

x

}

1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9