Application of Divide & Conquer :-

(i) Finding max & min

(ii) Binary search

(iii) Power of an element

(iv) Mergesort

(v) Quicksort

(vi) Strassen's matrix multiplication

(vii) counting inversion

(viii) selection procedure

(ix) Contiguous maximum subarray sum.

⇉ Finding Max & Min :-

I/p → array of elements

o/p → max & min element

eg:- A [25, 85, 115, 7, 99, 4, 96]
(1 2 3 4 5 6 7)

[Max = 115
Min = 4]

Steps :-

- check whether problem is small or big.

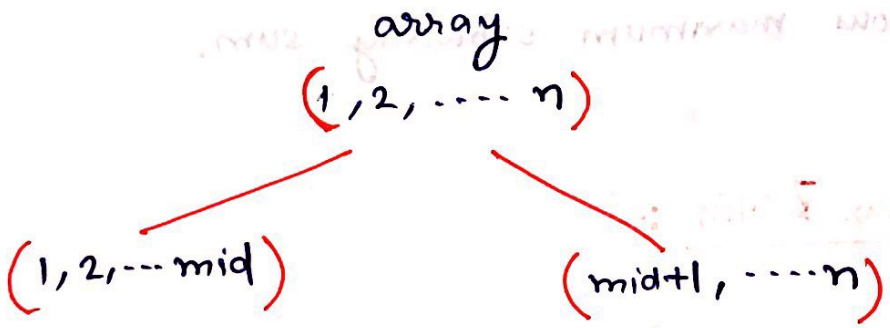
small problem – (1 or 2 element array)
n = 1 or 2

```
if ( problem = small)
    solve;
else
    divide;
```

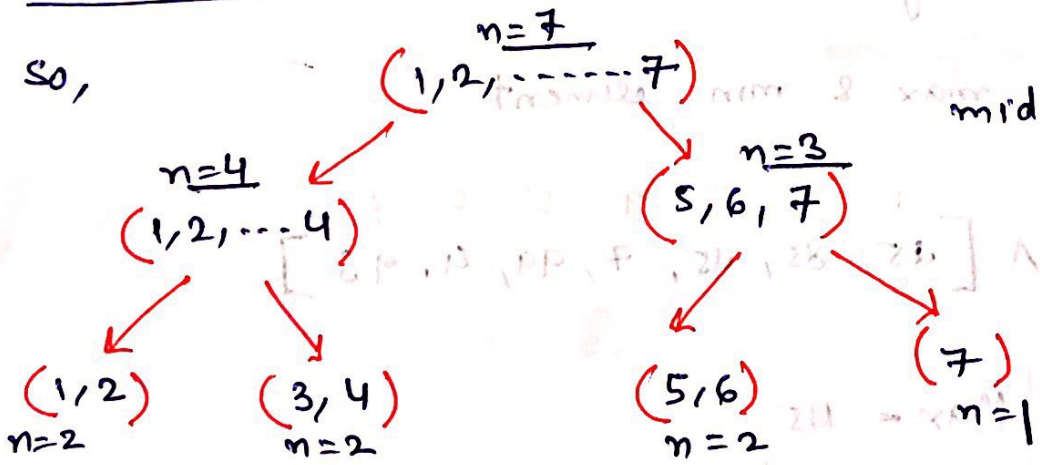
(n = last index − first index + 1)

- Divide :- Calculate mid value of array-

$$mid = \frac{\text{first index} + \text{last index}}{2}$$



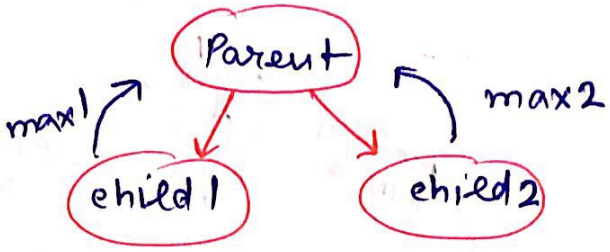
So,



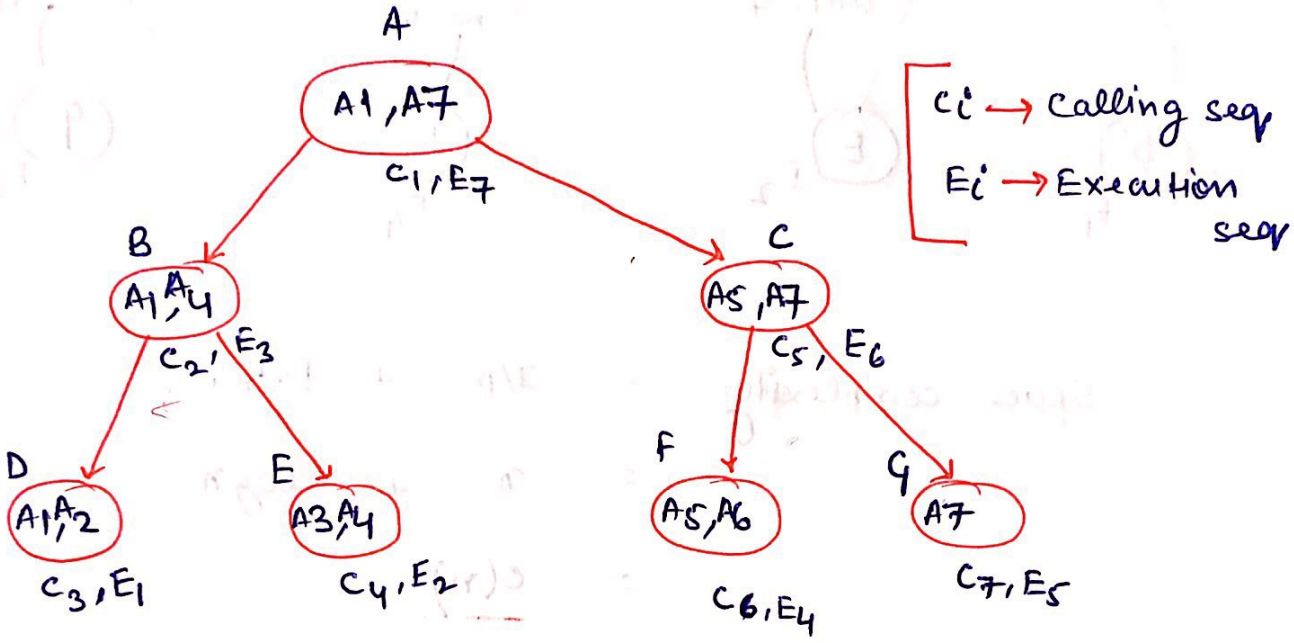
$mid = \frac{1+7}{2} = 4$

when n = 2 or 1 recursion stop

Conquier & Combine :-



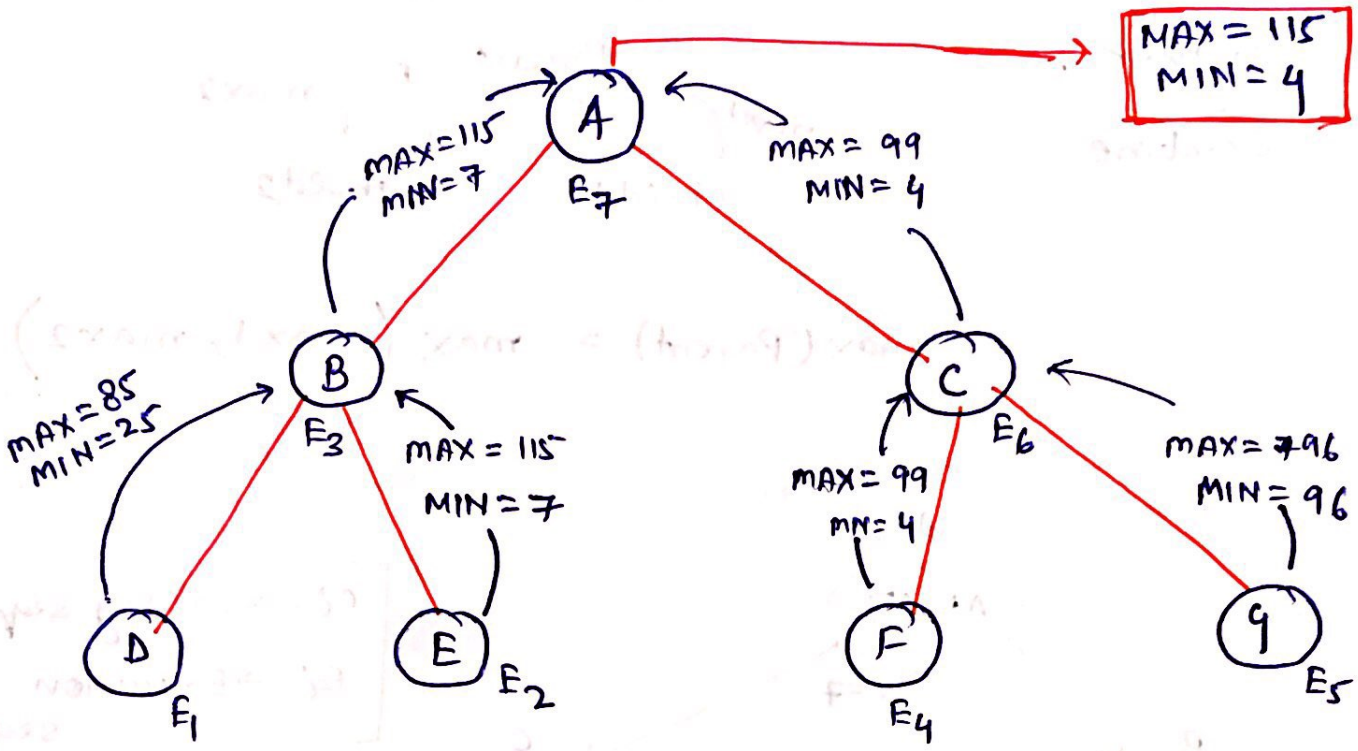
$$[\max(\text{Parent}) = \max(\max 1, \max 2)]$$



NOTE :- func^n Calling seq → Pre-Order (PUSH op^n)

func^n exec^n seq → Post-order (POP op^n)

1. PUSH C_1
2. PUSH C_2
3. PUSH C_3
4. POP C_3
5. PUSH C_4
6. POP C_4
7. POP C_2
8. PUSH C_5
9. PUSH C_6
10. POP C_6
11. PUSH C_7
12. POP C_7
13. ~~PUSH~~ POP C_5
14. POP C_1



Space complexity = I/p + Extra

= n + log n

= O(n)

Program :-

```
max_min ( a, i, j)        T(n)
{
    if ( i==j)
    {
        max = min = a[i];
    }
    else if ( i==j-1)
    {
        if ( a[i] < a[j])
            min = a[i] ; max = a[j];
        else
            min = a[j] ; max = a[i];
    }
```

single element sub-array

2 elements sub-array

```
else
{   mid = ⌊ (i+j)/2 ⌋ ;                                  # dividing
    (max1, min1) = max_min (a, i, mid) ;      T(n/2)
    (max2, min2) = max_min (a, mid+1, j) ;    T(n/2)
    if ( max1 < max2)
        max = max2 ;                                     # max calculation/
    else                                                   combining
        max = max1 ;
    if ( min1 < min2)                                    # min calculation/
        min = min1 ;                                       combining
    else
        min = min2 ;
}
```

- Let T(n) is time complexity of above program on n elements array.

Recurrence Relation for time complexity :-

$$T_n = \begin{cases} O(1) & \text{if } n=1, n=2 \\ \underbrace{O(1)}_{\text{divide}} + \underbrace{2T(n/2) + O(1)}_{\text{conquer (solving)}} & \text{if } n>2 \end{cases}$$

$$T(n) = 2T(n/2) + c$$

$$T(n) = O(n)$$

$$T(n) = 2T(n/2) + c$$

$$T(n/2) = 2T(n/4) + c$$

$$T(n/4) = 2T(n/8) + c$$

$$T(n/8) = 2T(n/16) + c$$

So, $T(n) = 2^k c(n/2^k) +$.

$= 4T(n/4) + 2c + c$

$= 8T(n/8) + 4c + 2c + c$

$= 16T(n/16) + 8c + 4c + 2c + c$

NOTE:- K is stack space so, K will denote space complexity

K times

$= 2^k T(n/2^k) + 2^{k-1}c + \cdots 2^2 c + 2^1 c + 2^0 c$

$n/2^k = 2$ 1, or 2 as $T(1)$ and $T(2) =$ const

$\Rightarrow K = \log_2 n$

$= n\,T(1) + c\left(2^0 + 2^1 + 2^2 + \cdots 2^{\log_2 n - 1}\right)$

$= n\,T(1) + c\left(\frac{1(2^{\log_2 n} - 1)}{2-1}\right)$

$= n$

$= O(n)$

- Let $T(n)$ is equal to no. of comparision to find max, min on n elements array, using above algorithm.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 0 + 2T(n/2) + 2 & \text{if } n > 2 \end{cases}$$

divide — conquer — combine

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

$$= 4T(n/4) + 4 + 2$$

$$= 8T(n/8) + 8 + 4 + 2$$

$$= 2^k T(n/2^k) + 2^k + \cdots 2^3 + 2^2 + 2^1$$

$n/2^k = 2$

$2^{k+1} = n$

$k = \log_2 n - 1$

$$= \frac{2^{\log_2 n}}{2} T(2) + 2^{\log_2 n - 1} + \cdots 2^3 + 2^2 + 2^1$$

$$= n/2\, T(2) + \frac{2(2^{\log_2 n - 1} - 1)}{2-1}$$

$$= n/2\, T(2) + 2(n/2 - 1)$$

$T(2)$ — (1)

$$= n/2 + n - 2$$

$$= \left[3n/2 - 2\right]$$ no. of comparision when $n > 2$

$$T(n) = O(n)$$

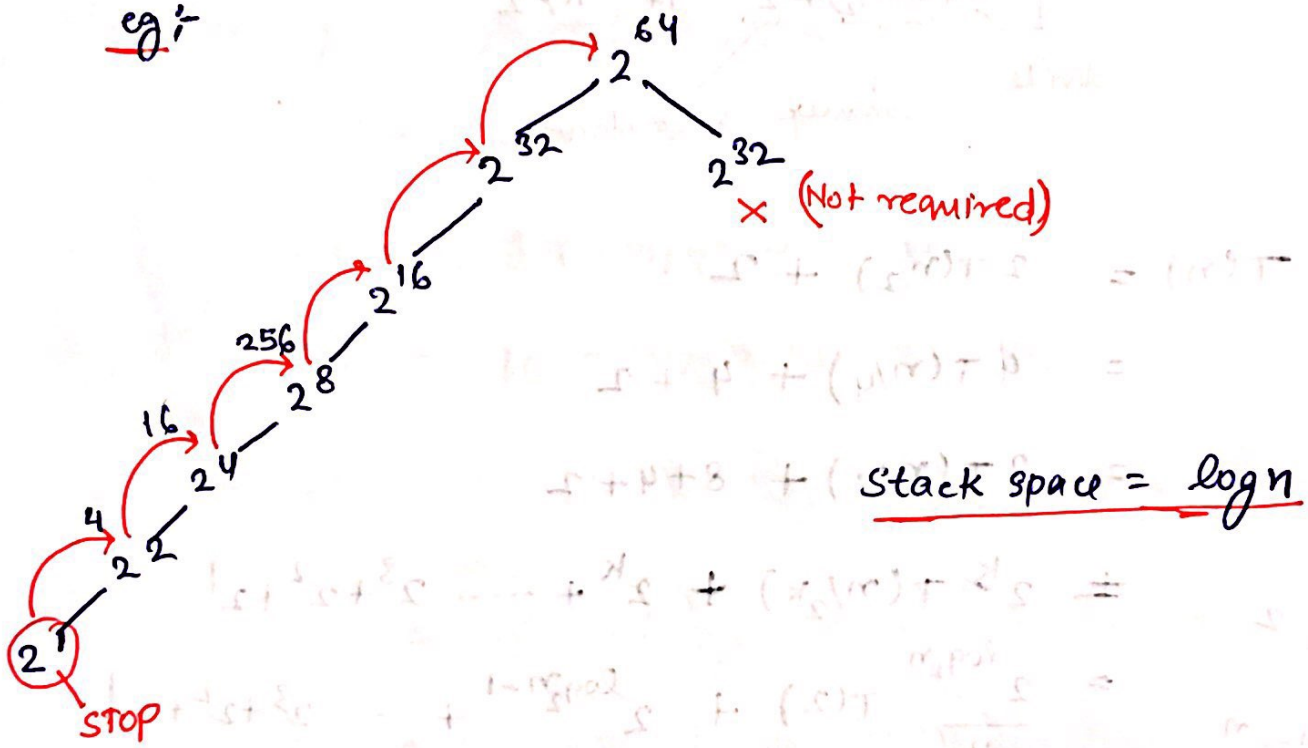
⇉ Power of an element :-

I/p → 2-integers $a \geq 2$, $n \geq 1$

O/p → a^n

$a^n = a^{n/2} * a^{n/2}$

eg:-



Stack space = log n

Program:-

```
T(n) power ( a, n)
{
     if ( n==1)
       return a;                    ] O(1)
     else
      {num = power ( a, n/2);       ] T(n/2)
       return (numxnum);            ] O(1)
      }                                 mid = n/2
                                        combine — O(1)
}
```

Let T_n = time complexity of above algorithm to find a^n,

Recurrence relation T_n :-

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

$T(n) = T(n/2) + c$ — divide time and combine time at top level of stack.

$= T(n/4) + 2c$

$= T(n/8) + 3c$

$= T(n/2^k) + kc$

$n/2^k = 1 \quad k = \log n$ — stack size

$= T(1) + \log n$

$T(n) = O(\log n)$ — $T(n)$ will be same for every case, there is no best or worst cases.

Space - complexity = I/p + Extra

$= O(1) + O(\log n)$

$= O(\log n)$

$T(n)$ = no. of multiplications for the above program.

Recurrence relation for $T(n)$ –

$$T(n) = \begin{cases} 0 & n=1 \\ T(n/2)+1 & n>1 \end{cases}$$

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

$$= T(n/4) + 1+1$$

$$= T(n/2^3) + 1+1+1$$

$$= T(1) + \log n$$

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

no. of multiplications = $\log n$

ALTERNATIVE :-

```
for (i=1 & to logn)
{
    s = s * s;
}
```

⇒ Binary Search :-

I/p :- An array of n elements (sorted)

O/p :- Find position of element x

else return (-1).

Eg:-

A [	10	20	30	40	50	60	70]
	1	2	3	4	5	6	7

$x = 40$

$mid = \frac{1+7}{2} = 4$ $\quad A[mid] == x$ then return mid; best case

NOTE:- BEST case & WORST case not depends upon I/p size, they depends on algorithm logic.

Program:-

```
Bsearch( a, p, q, x)        — T(n)
{
    if ( p == q)
    {
      if ( a[p] == x)        — O(1)
         return p;
      else
         return -1;
    }
```

```
else
{
    mid = ⌊(p+q)/2⌋;          — O(1)

    if( a[mid] == x)
      return mid;             — O(1) — c1

    else if ( a[mid] > x)
      return Bsearch ( a, p, mid-1, x);     T(n/2)

    else
      return Bsearch ( a, mid+1, q, x);     T(n/2)

}
}
```

Recurren relation for time complexity T(n) :-

$$T(n) = \begin{cases} O(1) & n=1 \\ c & n>1 \end{cases}$$ Best case

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

$$T(n) = T(n/2) + c$$

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

$$= T(n/2^3) + 3c$$

$$= T(n/2^k) + kc$$

$k \rightarrow \log n$ — stack space

$$T(n) = T(1) + \log n \cdot c$$

$$(T(n) = O(\log n)) \text{ — worst case}$$

	Best case	worst case	average case
Time complexity	$O(1)$	$O(\log n)$	$O(\log n)$

Q. A sorted array of n distinct integers is the I/p & O/p is to find any element $a[i]$ such that $a[i] = i$.

$O(\log n)$

$$A = \begin{bmatrix} -20 & -15 & -10 & \boxed{4} & 7 & 12 & 23 & 28 \\ 1 & 2 & 3 & \boxed{4} & 5 & 6 & 7 & 8 \end{bmatrix}$$

Best Case

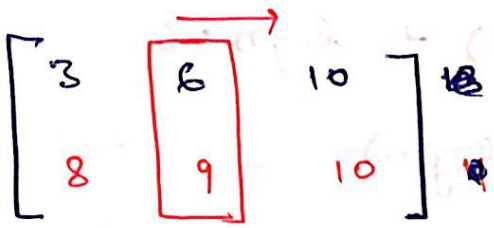
$$mid = \left\lfloor \frac{1+8}{2} \right\rfloor$$

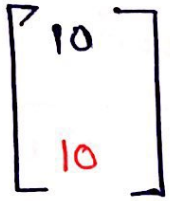
$$A = \begin{bmatrix} -45 & -40 & -31 & -25 & -20 & -10 & \boxed{1} & 3 & 6 & 10 & 13 & 18 & 21 & 37 \\ 1 & 2 & 3 & 4 & 5 & 6 & \boxed{7} & 8 & 9 & 10 & 11 & 12 & 13 & 14 \end{bmatrix}$$

$$mid = \left\lfloor \frac{1+14}{2} \right\rfloor$$

$$\begin{bmatrix} 3 & 6 & 10 & \boxed{13} & 18 & 21 & 37 \\ 8 & 9 & 10 & \boxed{11} & 12 & 13 & 14 \end{bmatrix}$$

$$mid = \frac{8+14}{2} = 11$$





Worst case

NOTE:- array should have distinct elements and sorted for applying binary search in this problem.

```
int search( a[], first, last)
{
    mid = (first + last)/2;
    if (a[mid] == mid)
        return mid;
    else if (a[mid] > mid)
        return search(a[], first, mid-1);
    else if (a[mid] < mid)
        return search(a[], mid+1, last);

    return -1;
}
```

Q. I/p — an array of n elements in which until some position all are integers and afterwards all are ∞.

O(log n)

O/p — Find the position of first ∞.

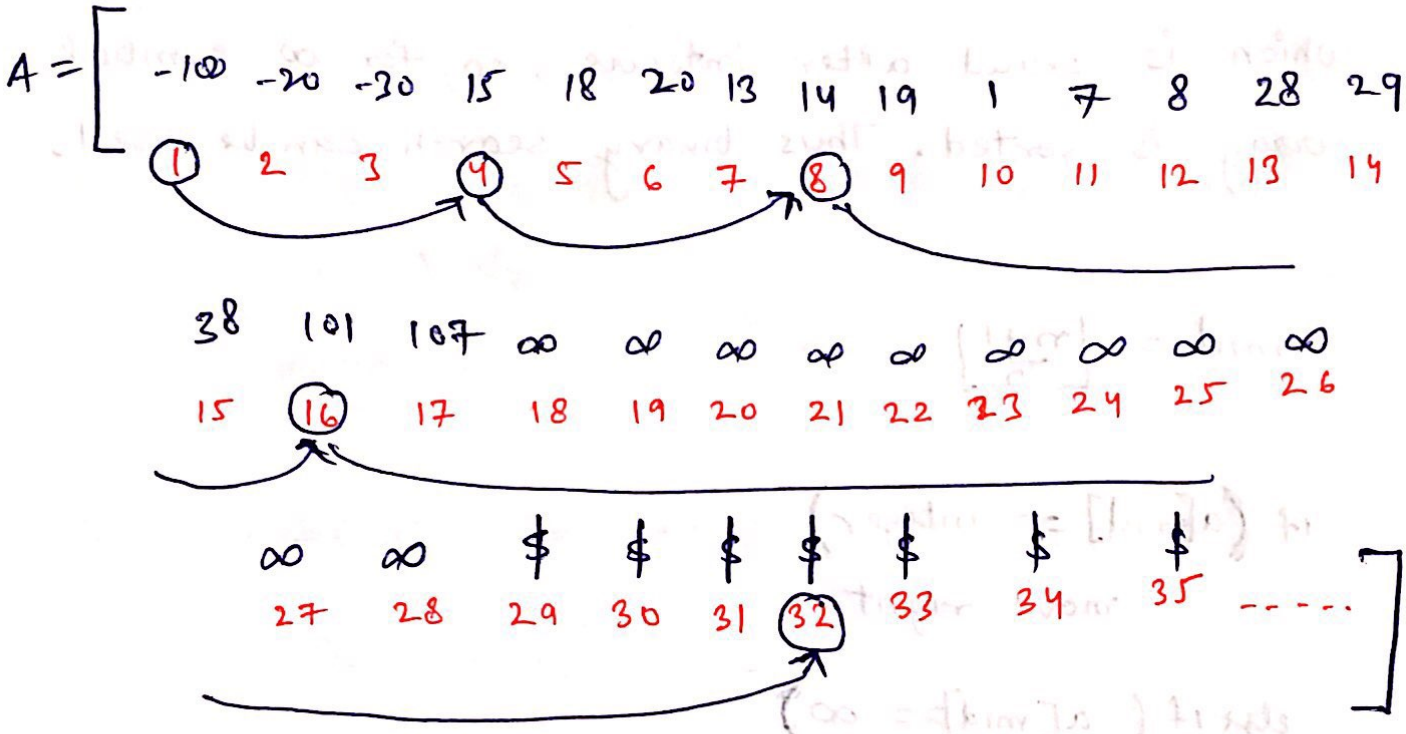
Since, here we have to search for a symbol ∞ which is placed after integers, so for ∞ symbol array is sorted. Thus binary search can be used.

$$mid = \left\lfloor \frac{n+1}{2} \right\rfloor$$

```
if (a[mid] == integer)
        move right;
else if ( a[mid] == ∞)
        if a[mid-1] == ∞
                move left;
        else
                return mid;
```

Q. An array of n-elements in which until some-position all i/p are ∞ (assume n is unknown & after the array all are $ symbol.

O/p – Find the position of 1st ∞.



- First Binary search applied for finding last index bound for array.

```
next = 1;
while ( a[next] ≠ $)
{   i=1;
    next = exp(2,i);      ⟹  next = 32
    i++;
}
return next;
```

- Now calculate mid = $\frac{1 + next}{2}$ and apply Binary search for ∞ symbol.

Q. I/p – A sorted array of n-integers

O/p – Find any two element a and b so that $a+b = 1000$.

$$A = \begin{bmatrix} 100 & 200 & 300 & 400 & 500 & 600 & 700 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

Here for each n elements apply Binary search

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

(i) n × Linear search (n) $n \times n$ $O(n^2)$

(ii) n × Binary search (n) $n \times \log n$ $O(n \log n)$

(iii) greedy algo n $O(n)$

greedy algo :-

(i) a = 1 , b = last

(ii) while (a != b)

```
if (a+b >==1000)
    return(a,b)
else if ( a+b>1000)
    b--;
else
    a++;
```

Q. I/p — A sorted array of n-integers

O/p — Find any 2-element a & b so that a+b > 1000

A [100 200 300 400 500 600 700]

1 2 3 4 5 6 7

return last two element (will have maximum sum)

a[last] + a[last-1]

T(n) = O(1)

NOTE:- In input array is not sorted then we have two algorithm :-

(i) sort array O(n log n)

(ii) return last two element O(1)

O(n log n)

(i) Find two max using two bubble pass =2n

O(n)