```
public static int fibonacci(int n) {
    if(n==0 || n==1) return n; ]⇒ O(1)

    fibNm1 = fibonacci(n-1);
    fibNm2 = fibonacci(n-2);
    fibN => fibNm1 + fibNm2; ] => O(1)
    return fibN;             ]⇒ O(1)
}
```

$f(0) = 0$

$f(1) = 1$

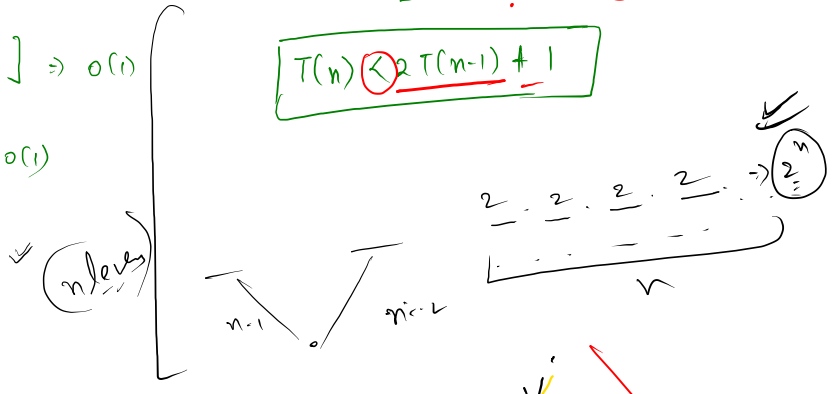
$T(n) = T(n-1) + T(n-2) + 1$ ①

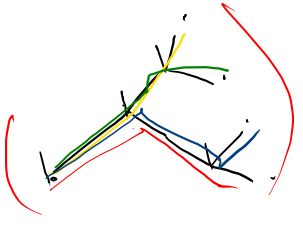
$(n-2) < (n-1)$

$T(n-2) < T(n-1)$ ↪ $4 < 5$, $T(4) < T(5)$

$T(n) = T(n-1) + T(n-2) + 1 < T(n-1) + T(n-1) + 1$

$T(n) < 2\,T(n-1) + 1$





$1 . 2 . 2 \Rightarrow 4$

$T(n) < 2\,T(n-1) + 1$

$2\,T(n-1) < 4\,T(n-2) + 2$

$4\,T(n-2) < 8\,T(n-3) + 4$

n terms

$T(1) =$

$T(n) = 1 + 2 + 4 + \ldots\ldots\ldots \Rightarrow 1\left(\frac{2^n - 1}{2-1}\right) \Rightarrow 2^n - 1$ (n terms)

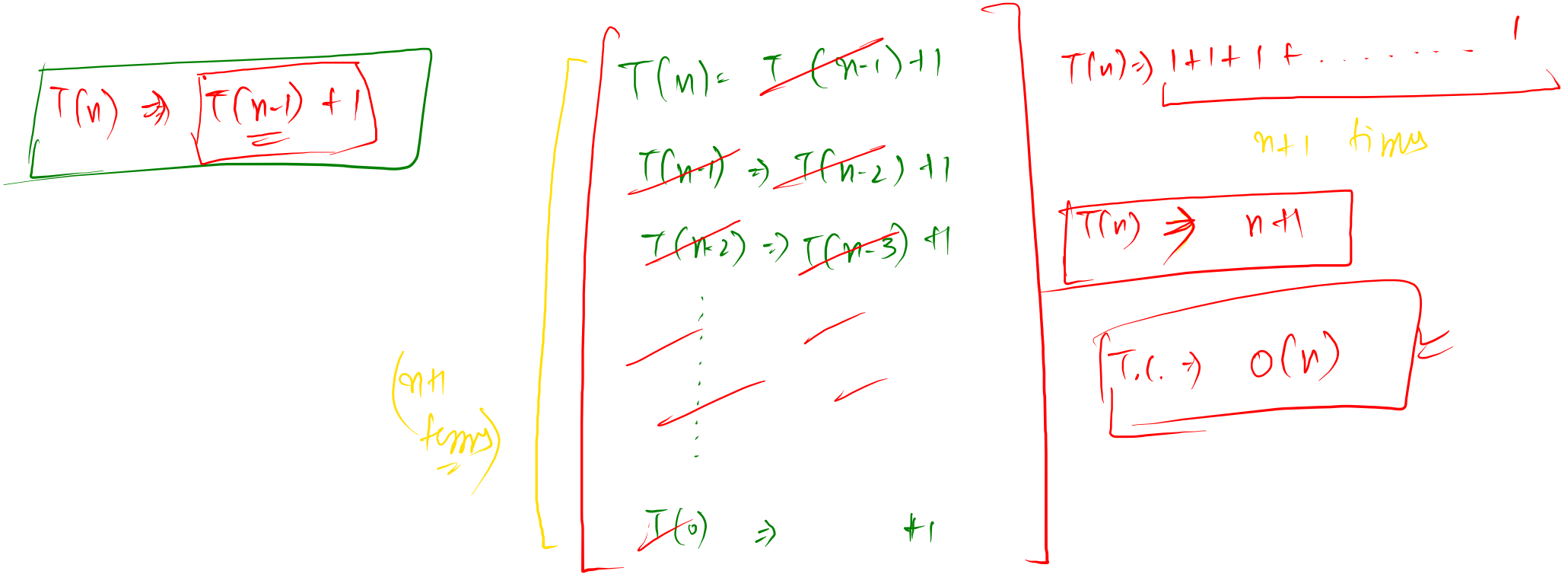
① Recurrence relation

② optimize

③ solve → substitution.

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

```java
public static void printIncreasing(int n) {
    if (n == 0) {
        return;
    }
    printIncreasing(n - 1);
    System.out.println(n);
}
```



```java
public static int powerLinear(int x, int n) {
    if (n == 0) {
        return 1;
    }
    int xPowNm1 = power(x, n - 1); // faith
    int xPowN = x * xPowNm1;
    return xPowN;
}
```

T(n) => T(n-1) + 1

T.C. => O(n)

```java
public static int powerLogarithmic(int x, int n) {
    if (n == 0) {
        return 1;
    }
    int xPowNb2 = power(x, n / 2);
    int ans = xPowNb2 * xPowNb2;
    if (n % 2 == 1) {
        ans = ans * x;
    }
    return ans;
}
```

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

$T(n/2) = T(n/4) + 1$

$T(n/4) = T(n/8) + 1$

K steps

$T(1) = T(0) + 1$

$T(0) = 1$

$T(n) \Rightarrow 1 + 1 + 1 + \dots + 1$ (K times)

$T(n) \Rightarrow \log_2 n + 2$

T.C. $\Rightarrow O(\log_2 n)$

$\frac{n}{2^{K-2}} = 1$

$(n = 2^{K-2})$

$\log_2 n = K - 2$

$\log_2 n + 2 = K$

$n/2^0$ n

$n/2^1$ $n/2$

$n/2^2$ $n/4$

$n/2^{K-2}$ 1

$0/2^{K-1}$ 0

```java
public static void toh(int n, int src, int dest, int helper) {
    if (n == 0) {
        return;
    }
    toh(n - 1, src, helper, dest);
    System.out.println(n + "[" + src + " -> " + dest + "]");
    toh(n - 1, helper, dest, src);
}
```

K steps

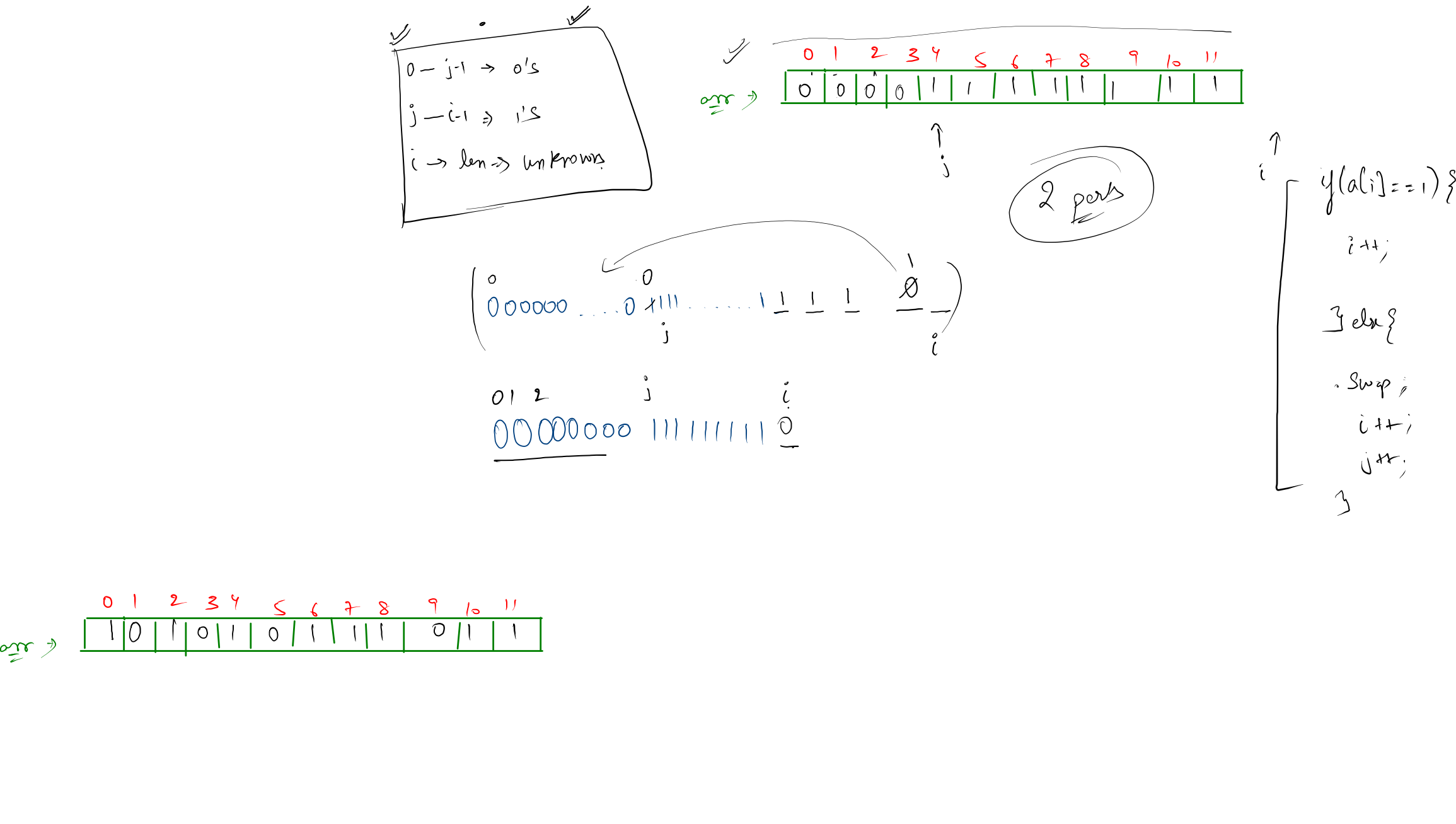
$T(n) = 2T(n-1) + 1$

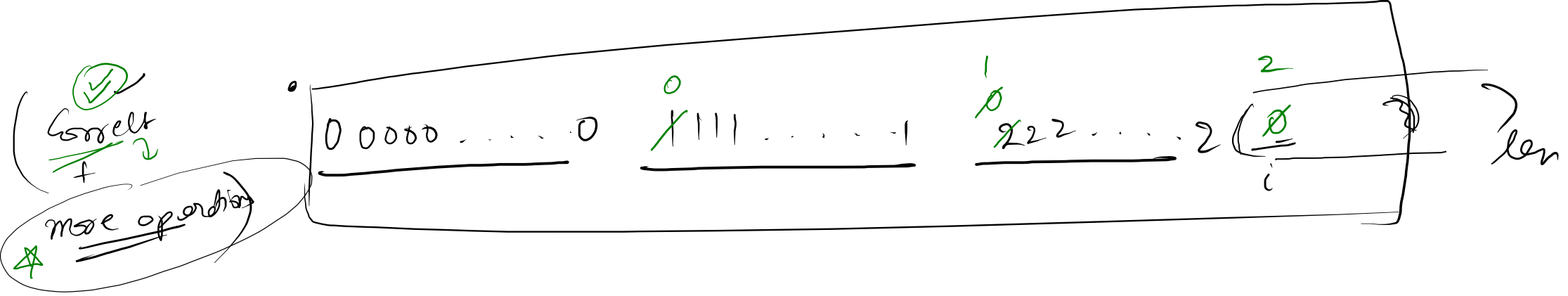
$T(n) = 2^n - 1$

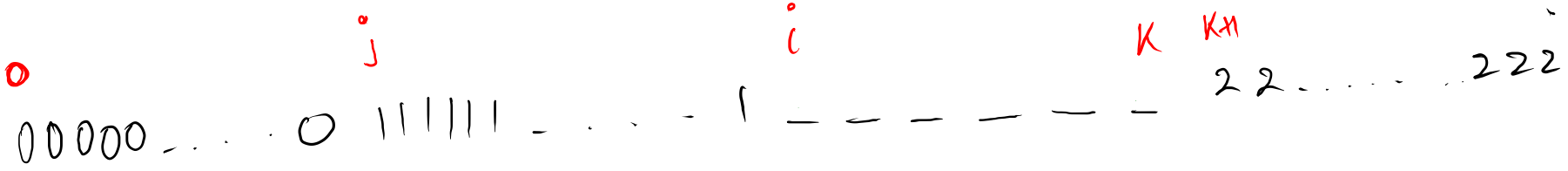
T.C. - $O(2^n)$

```java
public static void pzz(int n) {
    if (n == 0) {
        return;
    }
    System.out.print(n + " ");
    pzz(n - 1);
    System.out.print(n + " ");
    pzz(n - 1);
    System.out.print(n + " ");
}
```

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


0 – j-1 → 0's
j – i-1 ⇒ 1's
i → len → unknown
arr
0 1 2 3 4 5 6 7 8 9 10 11
0 0 0 0 1 1 1 1 1 1 1 1
j
2 pers
i
if(a[i]==1){
i++;
}else{
swap;
i++;
j++;
}
0 0 1
0 0 0 0 0 0 1 1 1 1
j i
0 1 2 j i
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0
arr
0 1 2 3 4 5 6 7 8 9 10 11
1 0 1 0 1 0 1 1 1 0 1 1





$0-(j-1)$: 0's

$j-(i-1)$: 1's

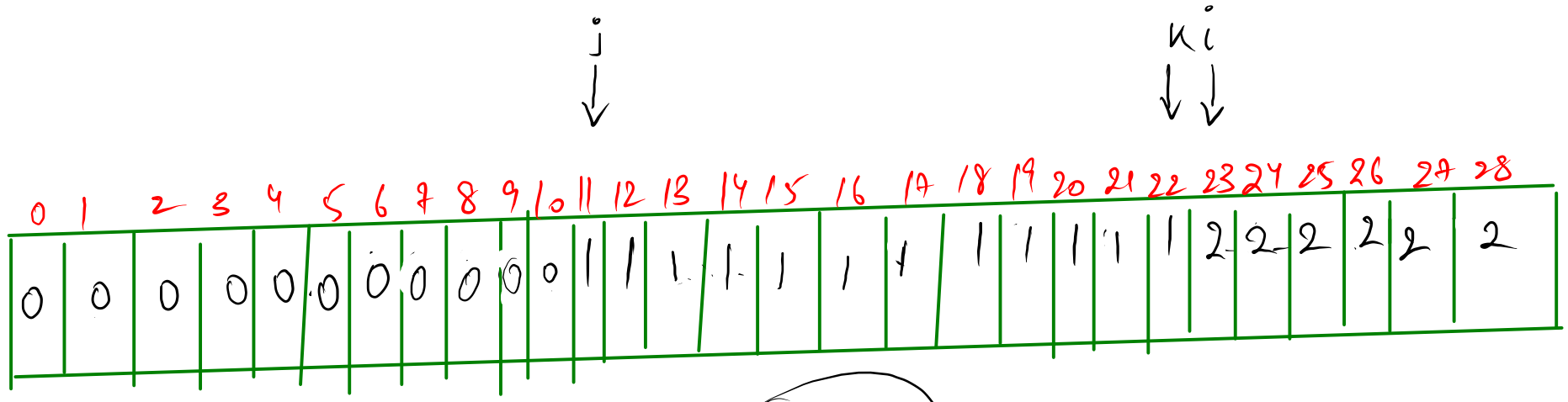
$(i-K)$: unknown (0/1/2)

$(K+1)-(len-1)$: 2's

a[i] = 1
↳ i++

a[i] = 0
↳ swap(i, j)
↳ i++, j++

a[i] = 2
↳ swap(i, K)
↳ K--



i > k ⇒ Complete

a[i] = 1
↳ i++

a[i] = 0
↳ swap(i, j)
↳ i++, j++

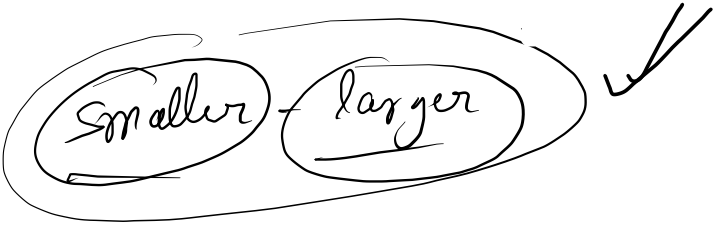
a[i] = 2
↳ swap(i, k)
↳ k--

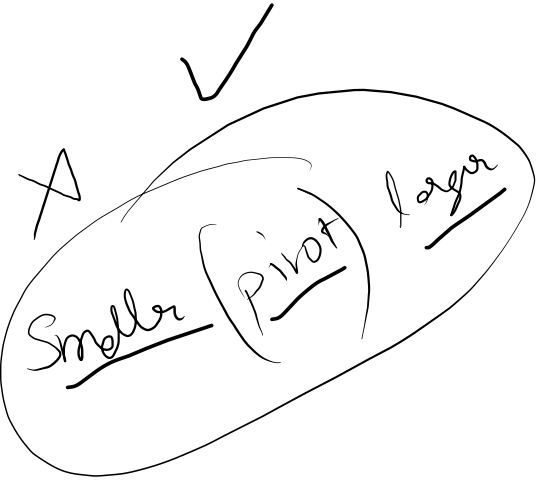
0 – (j-1) : 0's

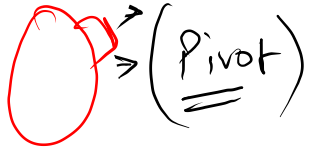
j – (i-1) : 1's

(i – k) : unknown (0/1/2)

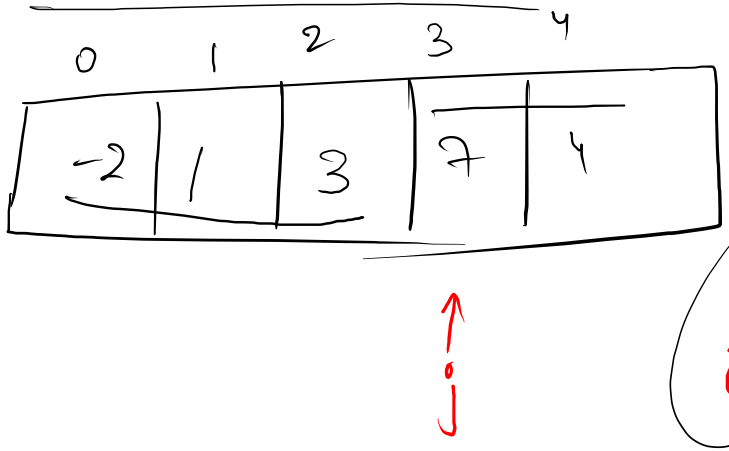
(k+1) – (len-1) : 2's







Smaller ⇒ a[i] <= Pivot
larger ⇒ a[i] > Pivot

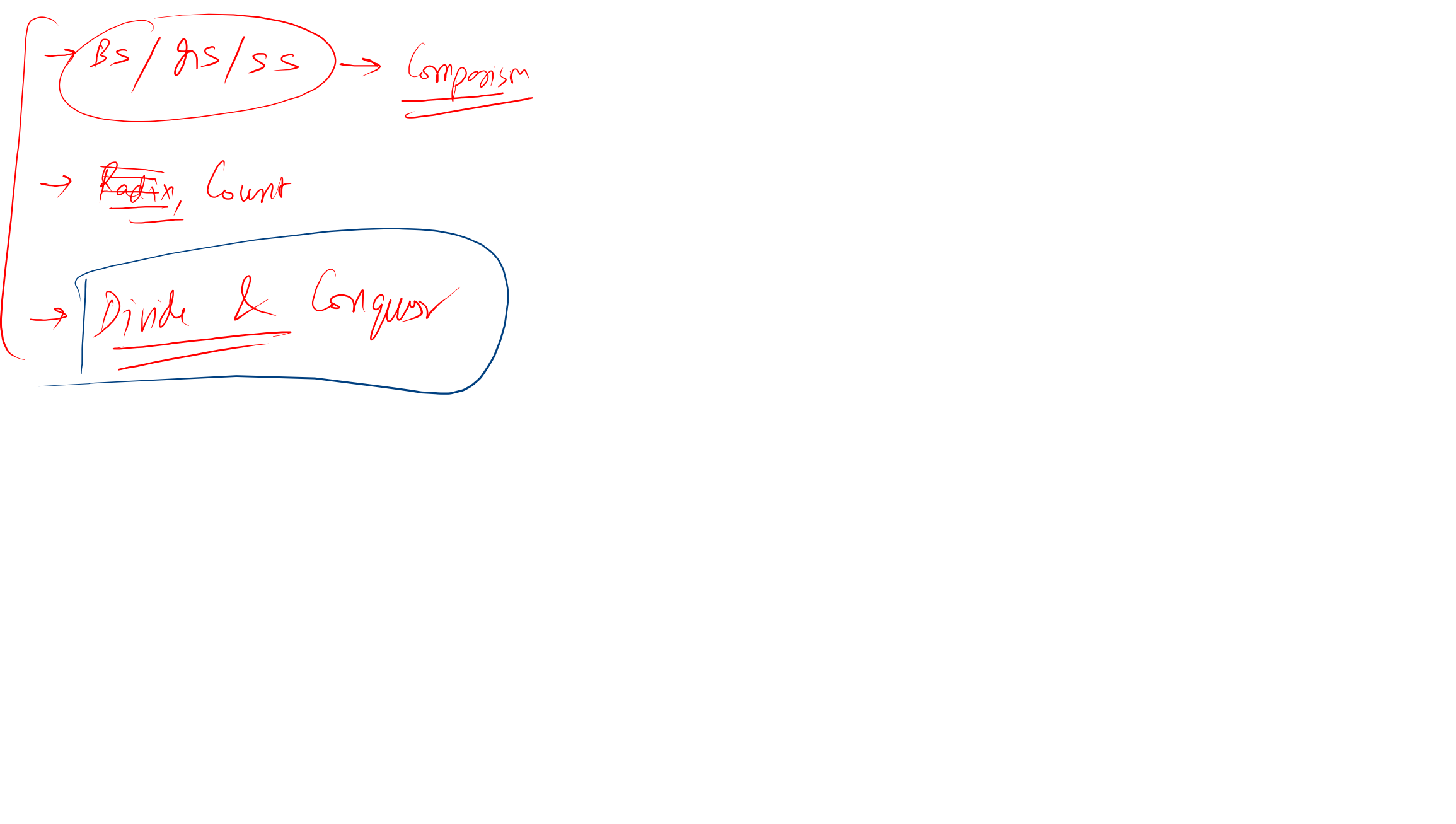


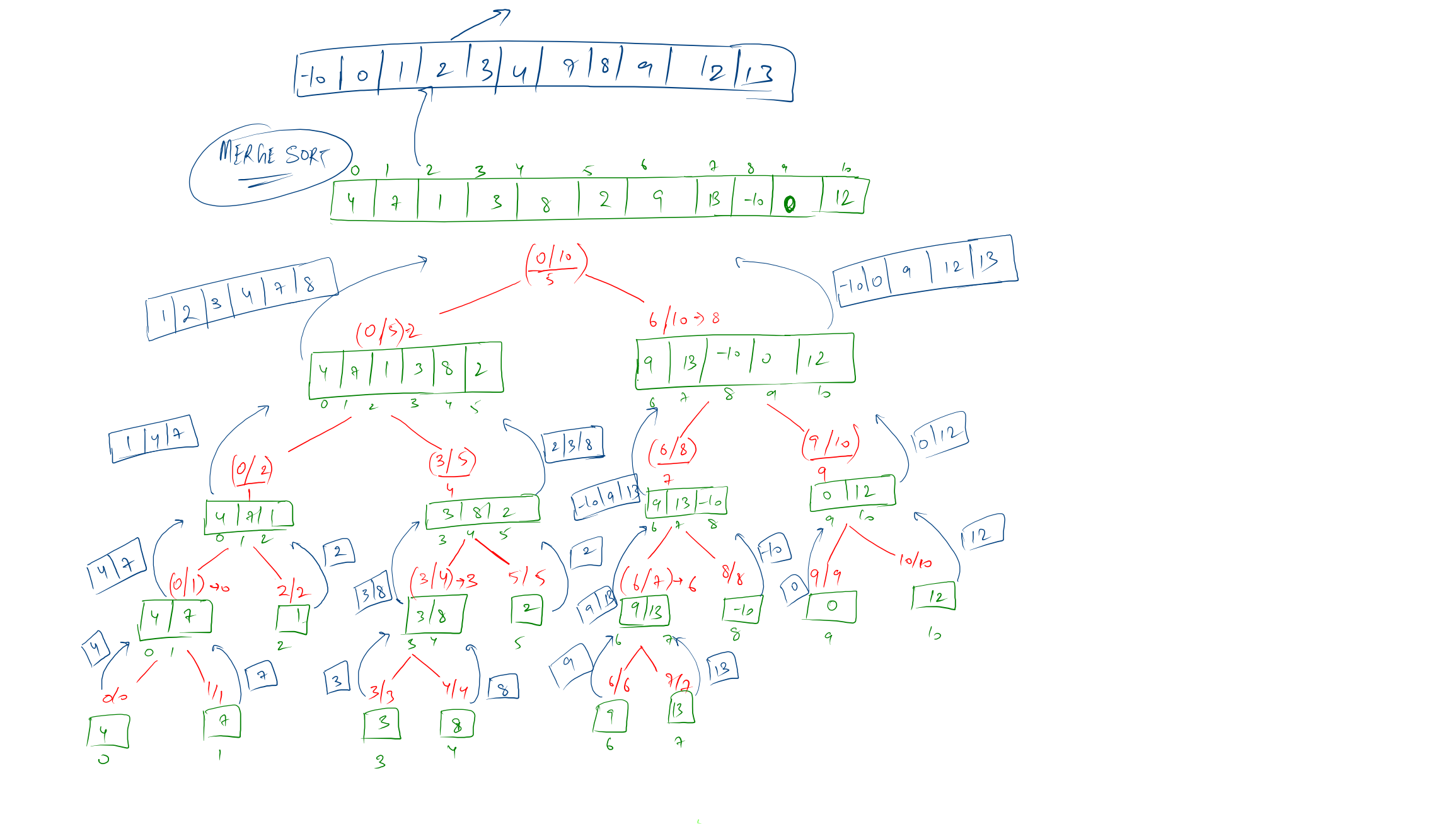
0 j i
Smaller larger K

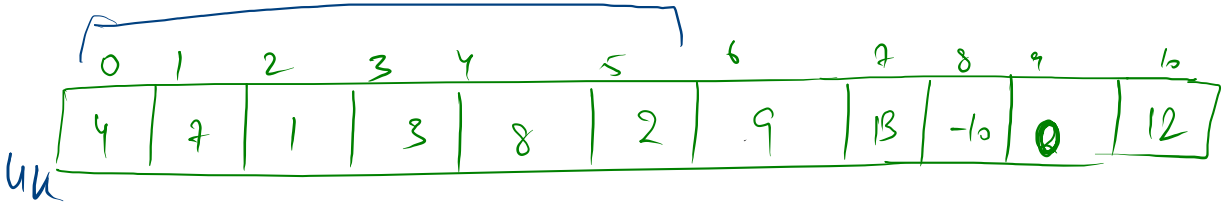
0 - (j-1) ⇒ Smaller

j - (i-1) ⇒ larger

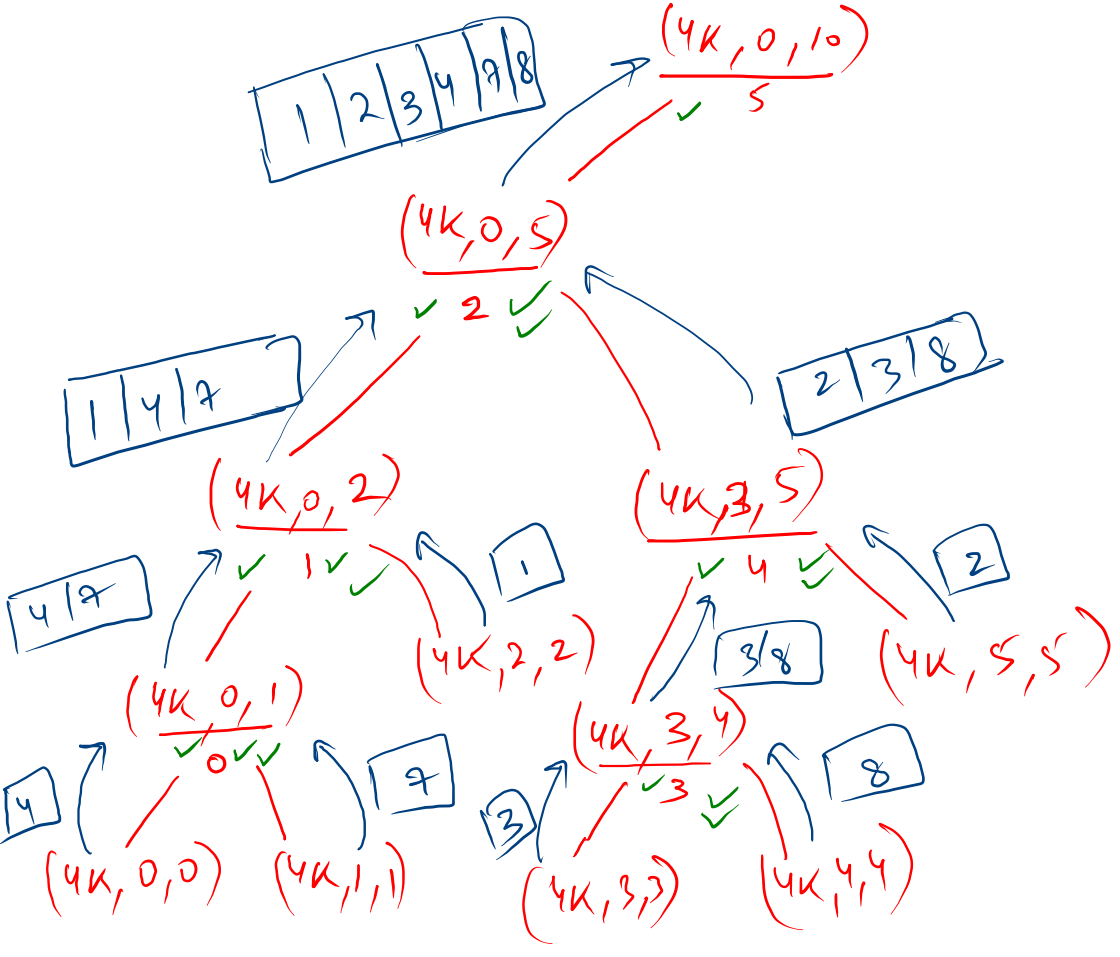
i - (len-1) ⇒ unKnowns


BS / MS / SS
Comparism
Radix, Count
Divide & Conquer


MERGE SORT




```java
public static int[] mergeSort(int[] arr, int lo, int hi) {
  if(lo == hi){
      return new int[]{arr[lo]};
  }
  int mid = (lo + hi) / 2;
  int lpart[] = mergeSort(arr,lo,mid); -①
  int rpart[] = mergeSort(arr,mid+1,hi); -②
  return mergeTwoSortedArrays(lpart,rpart); -③
}
```



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

↳ Recurrence Relation

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

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

$4 \cdot T(n/4) = 8 \cdot T(n/8) + n$

$\vdots$

$T(1) =$

$T(n) \Rightarrow \underbrace{n + n + n + \dots}_{k \text{ times}} \Rightarrow n \cdot k$ — (I)

k terms

$n \rightarrow n/2^0$

$n/2 \rightarrow n/2^1$

$n/4 \rightarrow n/2^2$

$\vdots$

$1 \rightarrow n/2^{k-1}$

$1 = \frac{n}{2^{k-1}}$

$2^{k-1} = n$

$k - 1 = \log_2^n$

$k = \log_2^n + 1$ — (II)

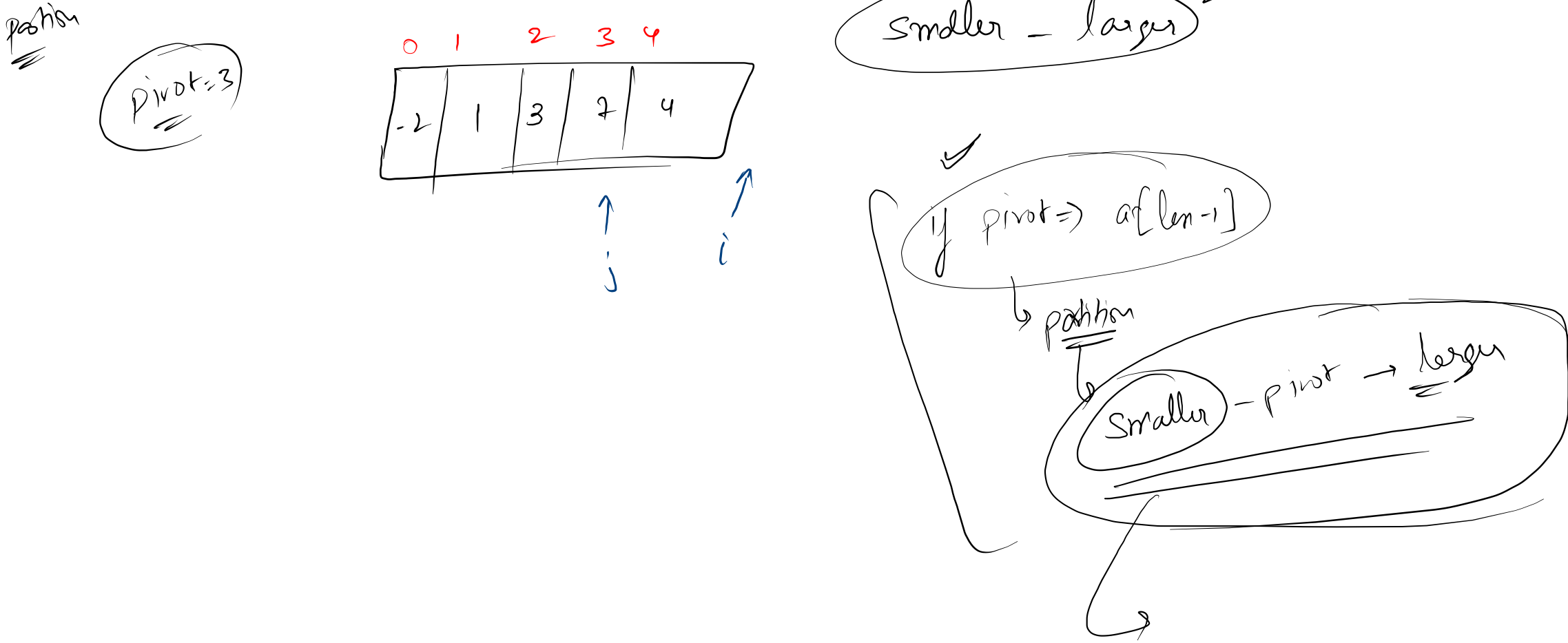
from (I) & (II)

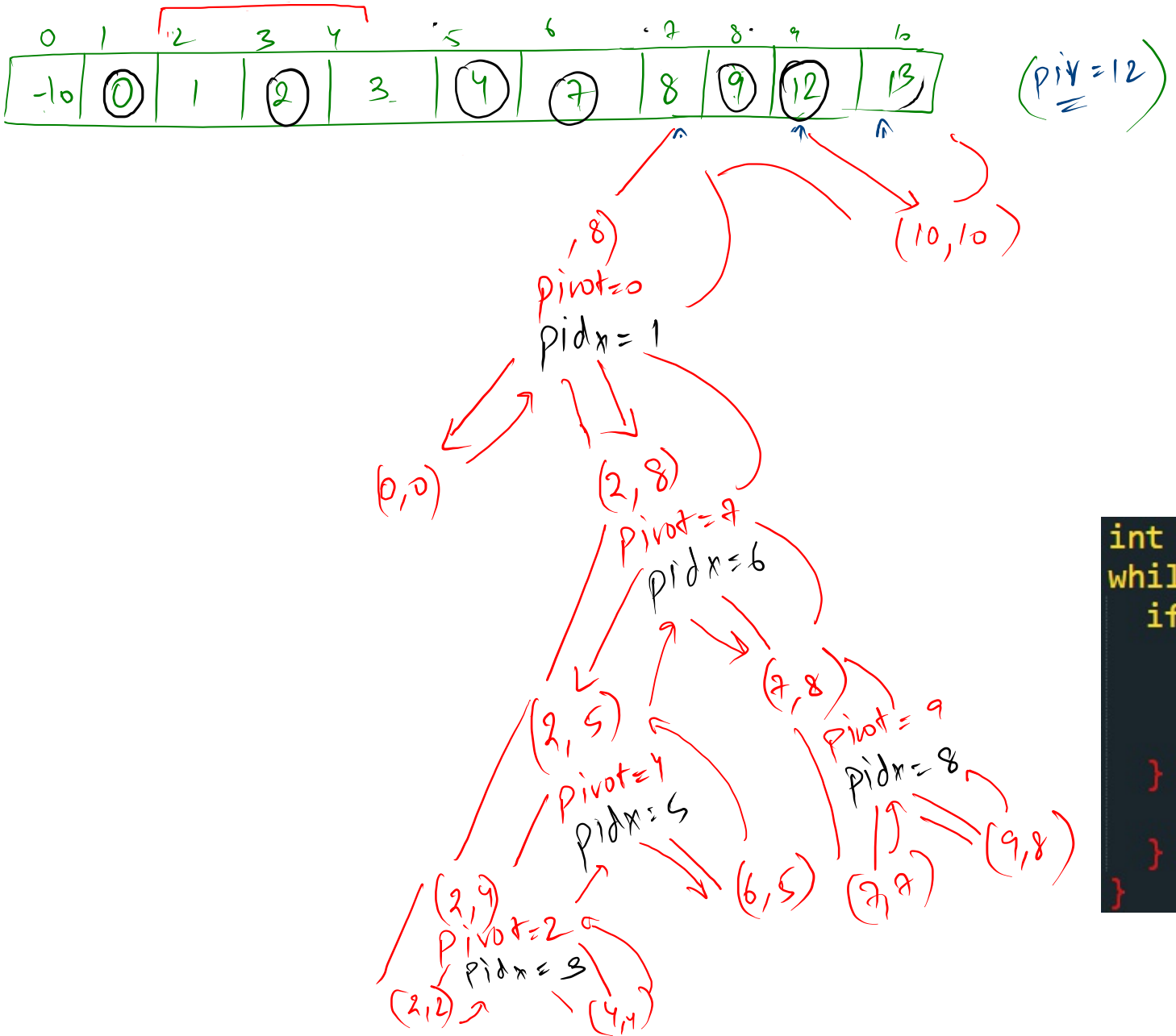
$T(n) \Rightarrow n[\log_2^n + 1]$

$T(n) \Rightarrow n \log_2^n + n$

T.C. $\Rightarrow O(n \log_2^n)$

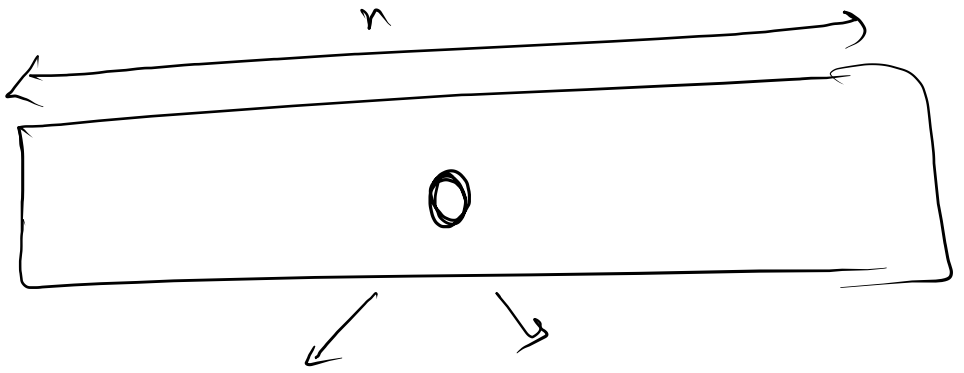
(Space Complexity $\Rightarrow O(n)$)


partition
pivot=3
0 1 2 3 4
-2 1 3 7 4
j
i
smaller - larger
if pivot => a[len-1]
partition
smaller - pivot → larger




```java
int i = lo, j = lo;
while (i <= hi) {
  if (arr[i] <= pivot) {
    swap(arr, i, j);
    i++;
    j++;
  } else {
    i++;
  }
}
```

~~Array~~ / Bonsai



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

10:05

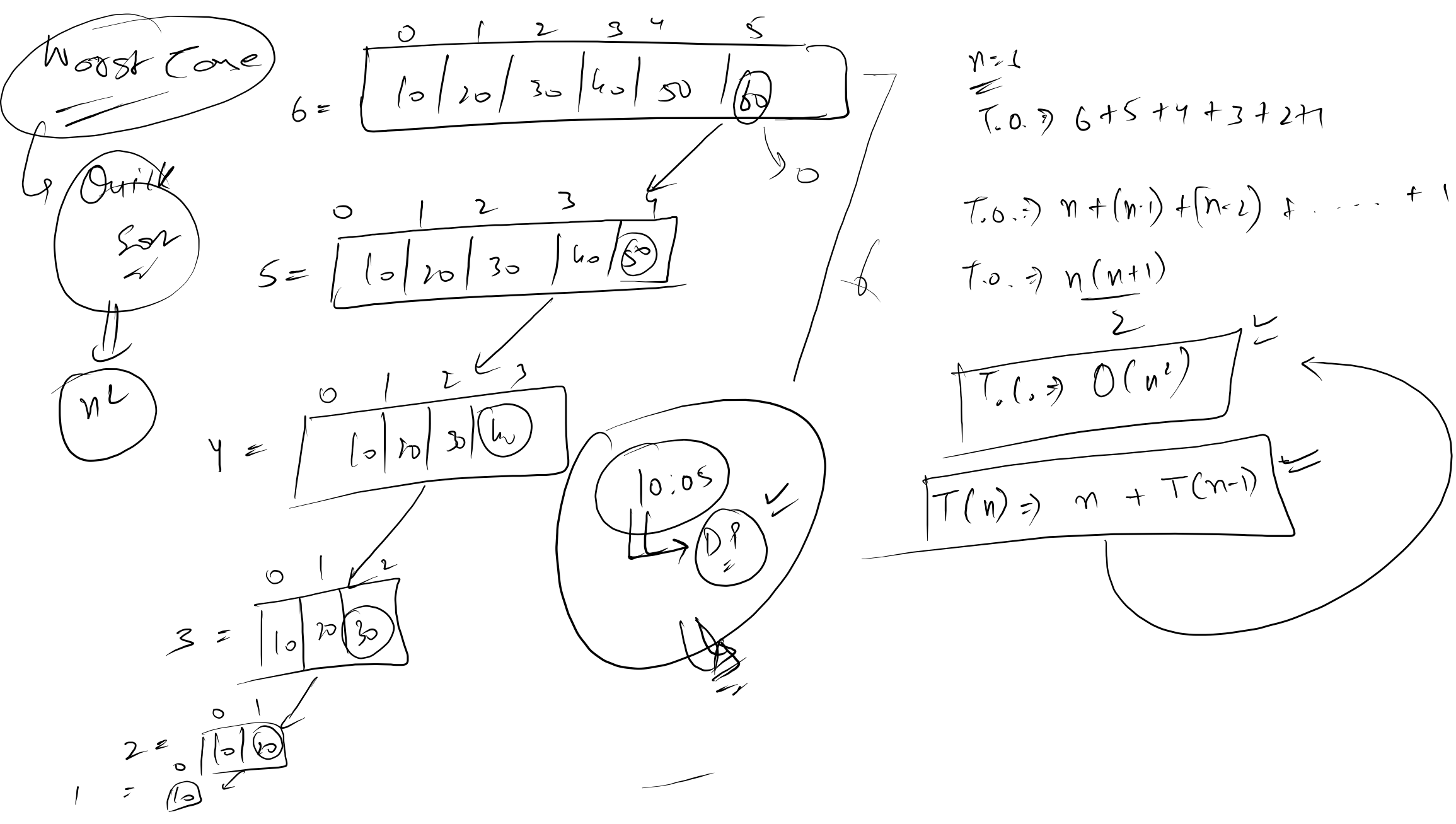
0 → array

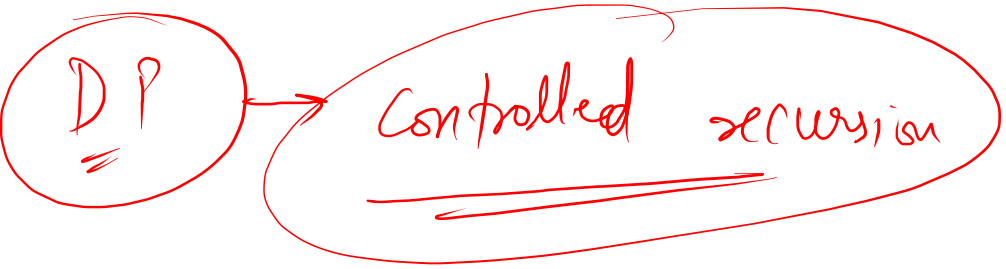
~~T=O(n)~~

$T(n) = n \log_2 n + n$

$T.C. \Rightarrow \Omega(n \log \frac{n}{2})$

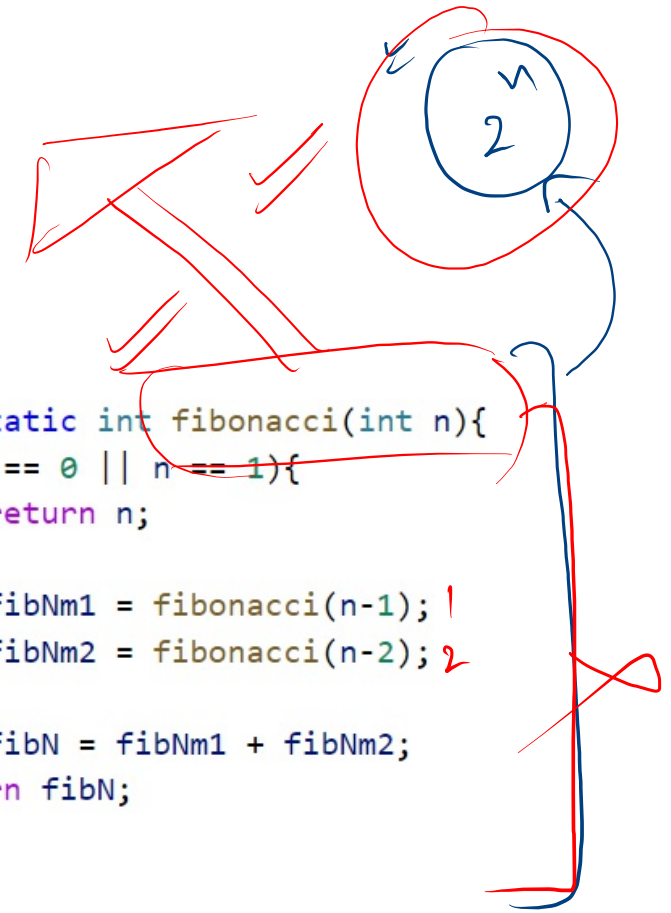
$S.C. \Rightarrow \Omega(1)$


Worst Case
Quick Sort
n²
6 = 10 20 30 40 50 60
5 = 10 20 30 40 50
4 = 10 20 30 40
3 = 10 20 30
2 = 10 20
1 = 10
10:05
DP
n=1
T.O. ⇒ 6+5+4+3+2+1
T.O. ⇒ n+(n-1)+(n-2)+ .... +1
T.O. ⇒ n(n+1)/2
T.C. ⇒ O(n²)
T(n) ⇒ n + T(n-1)

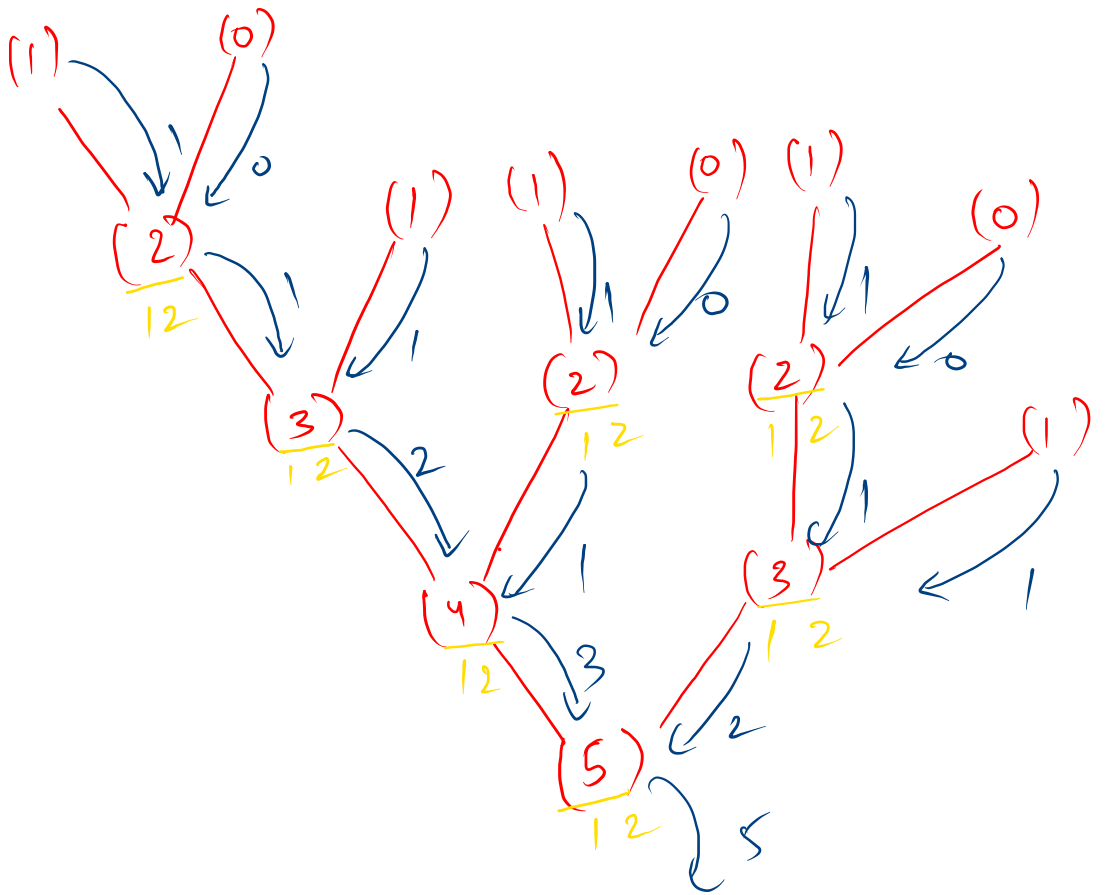

DP
Controlled recursion

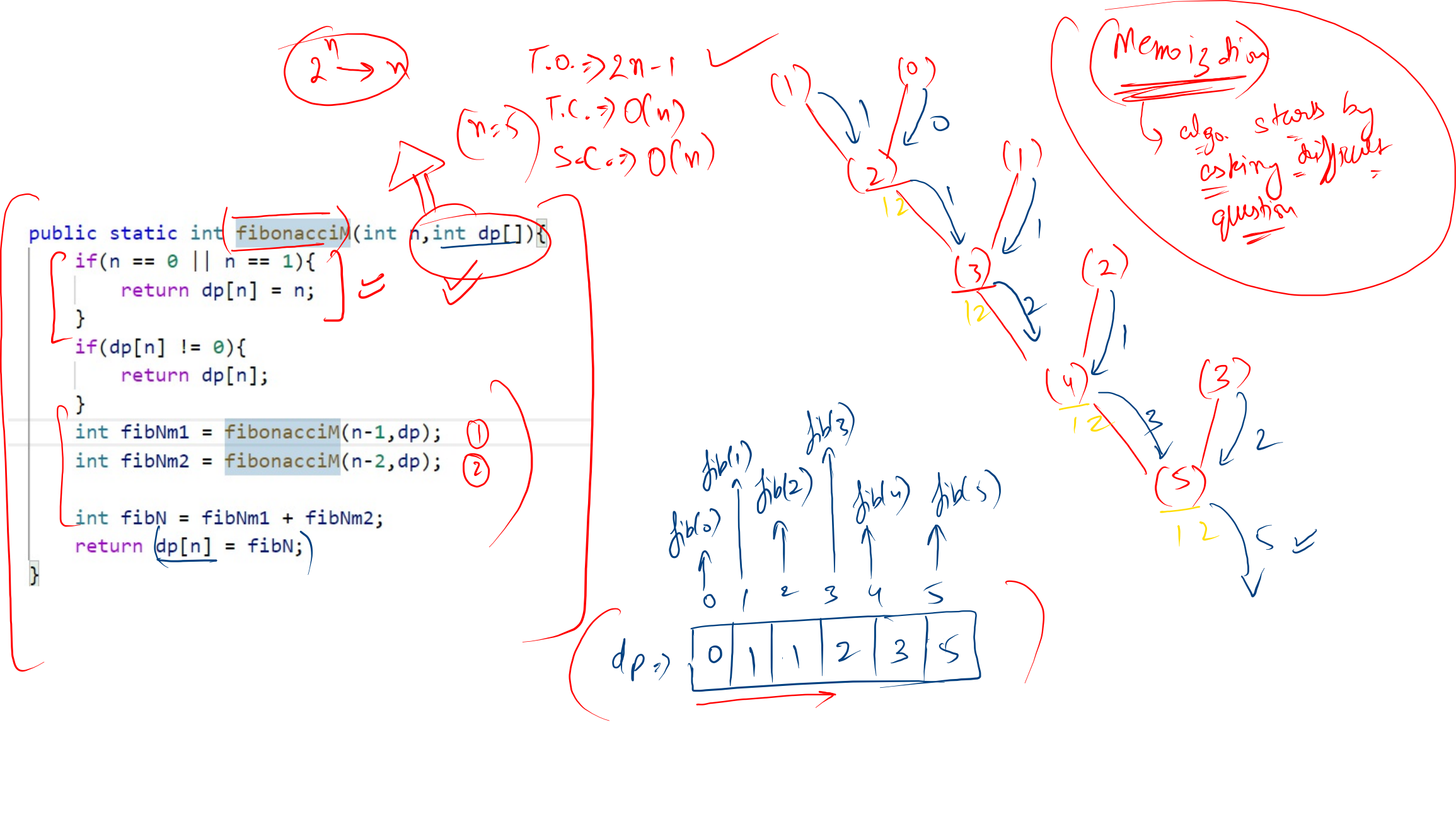
Overlapping Subproblem

n=5 → 5, 4, 3, 2, 1, 0 → nth

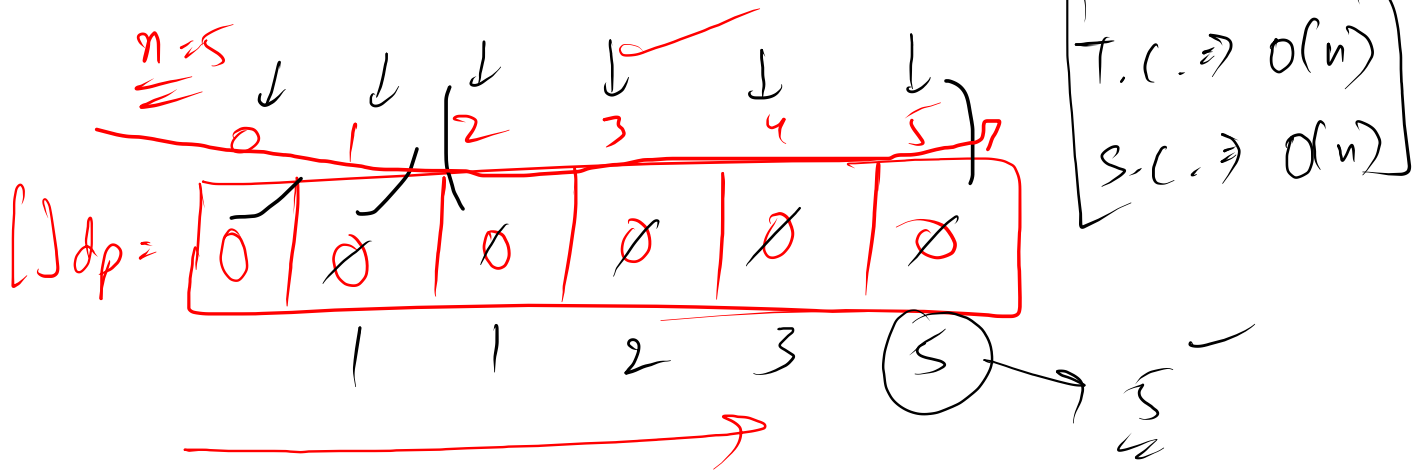


```java
public static int fibonacci(int n){
    if(n == 0 || n == 1){
        return n;
    }
    int fibNm1 = fibonacci(n-1);
    int fibNm2 = fibonacci(n-2);

    int fibN = fibNm1 + fibNm2;
    return fibN;
}
```




2^n → n
T.C. => 2n-1
(n=5) T.C. => O(n)
S.C. => O(n)
Memoization
algo. starts by asking difficult question
public static int fibonacciM(int n,int dp[]){
if(n == 0 || n == 1){
return dp[n] = n;
}
if(dp[n] != 0){
return dp[n];
}
int fibNm1 = fibonacciM(n-1,dp); (1)
int fibNm2 = fibonacciM(n-2,dp); (2)
int fibN = fibNm1 + fibNm2;
return dp[n] = fibN;
}
fib(0) fib(1) fib(2) fib(3) fib(4) fib(5)
0 1 2 3 4 5
dp => 0 1 1 2 3 5

```java
public static int fibonacciT(int n){
    int dp[] = new int[n+1];
    for(int i = 0 ; i <= n ; i++){
        if(i == 0){
            dp[0] = 0;// fib(0) = 0
        }else if(i == 1){
            dp[1] = 1;// fib(1) = 1
        }else{
            dp[i] = dp[i-1] + dp[i-2];
        }
    }

    return dp[n];
}
```



T.C. → O(n)

S.C. → O(n)

Tabulation → algo starts by solving easiest question & builds upto complex one.

```java
public static int fibonacciTOptimized(int n){
    if(n == 0 || n == 1){
        return n;
    }
    int f = 0 , s = 1;
    for(int i = 2 ; i <= n ; i++){
        int t = f + s;
        f = s;
        s = t;
    }

    return s;
}
```