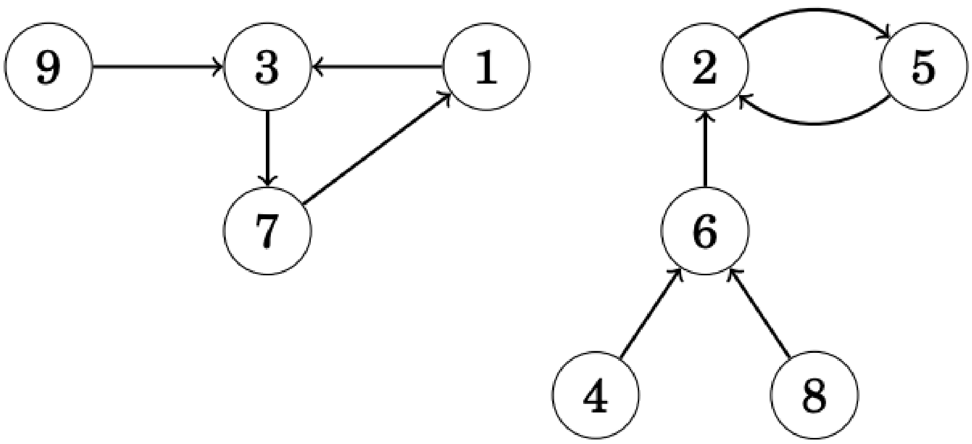
Kth Node in a Successor Graph

Successor graphs are those graphs where the **outdegree** of each node is 1, i.e., exactly one edge starts at each node. A successor graph consists of one or more components, each of which contains **one cycle** and some paths that lead to it.
Successor graphs are sometimes called functional graphs. The reason for this is that any successor graph corresponds to a function that defines the edges of the graph. The parameter for the function is a node of the graph, and the function gives the successor of that node.

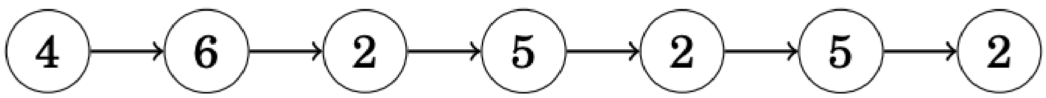
For example, the function

x	: 1 2 3 4 5 6 7 8 9
succ(x):	3 5 7 6 2 2 1 6 3

defines the following graph:



Since each node of a successor graph has a unique successor, we can also define a function succ(x, k) that gives the node that we will reach if we begin at node x and walk k steps forward. For example, in the above graph succ(4, 6) = 2, because we will reach node 2 by walking 6 steps from node 4:



So our problem is that we are given a node x and an integer k, we want to efficiently process the queries of the form succ(x, k).

Brute force
A straightforward way to calculate a value of succ(x, k) is to start at node x and walk k steps forward, which takes O(k) time.

Here is the simple implementation of the above idea.

```
/*
	Function to find the Kth successor of node x
*/
function succ(x, k, successor)

	//   repeat k times, change x -> successor(x)
	for i from 1 to k
		x = successor(x)

	return x
```

Time Complexity: O(K) per query, where 'K' is the number of successors we are interested in.

Space Complexity: O(1), since we are not using any additional space to find the kth successor of a node.

Dynamic Programming :
We will use a crucial observation to apply dynamic programming concepts here.

Note that the path of length K to the Kth successor can be broken down into logK paths each with a length of power of two (This is simply the binary representation of K).

This is because the Kth successor of a node is the bth successor of the ath successor of the node where:

$$a + b = K$$

So we will pre-calculate the kth successors of every node (where k is a power of 2) and finally calculate the required value by joining these paths.

An example will make the picture more clear.

For example:
If we want to calculate the value of succ(x, 11), we first form the representation

$$11 = 8 + 2 + 1$$

Using that,

$$succ(x,11) = succ(succ(succ(x,8),2),1).$$

For example, in the previous graph
succ(4, 11) = succ(succ(succ(4, 8), 2), 1) = 5.

So the 11th successor of node 4 is just the 8th successor of the 3rd successor of node 4.

Similarly the 3rd successor of node 4 is the 2nd successor of the 1st successor of node 4.

Using preprocessing, any value of succ(x, k) can be calculated in only **O(logk)** time.
The idea is to precalculate all values of succ(x, k) where k is a power of two and is at most equal to **2^(MaxLog)**, where **MaxLog** is the maximum value of log(K) across all queries.

For this we will break down the powers into half and solve for smaller subproblems.
This is because

k = (k/2) + (k/2)

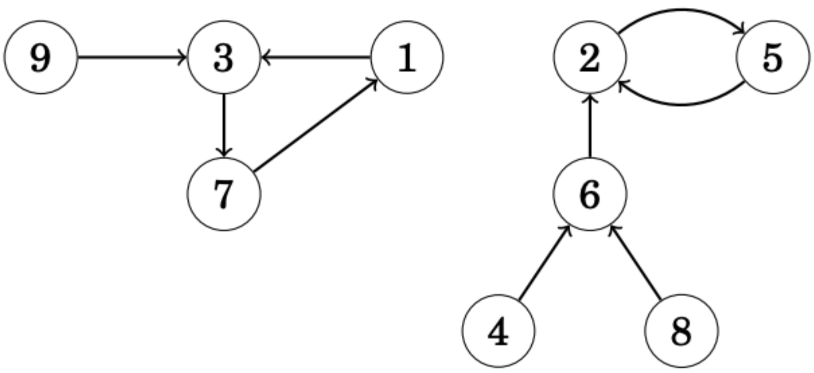
and using the above property we can half the value of k every time. So this gives us the following recurrence relation:-

succ(x, k) =

{ successor(x), k = 1
succ(succ(x, k/2), k/2), k > 1
}

Precalculating the values takes O(n*MaxLog) time, because O(MaxLog) values are calculated for each node.

In the above graph, the first values are as follows:



x	1	2	3	4	5	6	7	8	9
succ(x,1)	3	5	7	6	2	2	1	6	3
succ(x,2)	7	2	1	2	5	5	3	2	7
succ(x,4)	3	2	7	2	5	5	1	2	3
succ(x,8)	7	2	1	2	5	5	3	2	7

succ(x, 1) will be the base case and those values will be provided to us in the input.
We will then use the values of succ(y, 2^{j-1}) to calculate the value of succ(x, 2^j) for some values of y and x.

After this, any value of succ(x, k) can be calculated by presenting the number of steps k as a sum of powers of two. This technique is also called **Binary lifting**.

The **pseudo code** is as follows.

```
/*
        Function to calculate the dp table for each node from 1 to n
*/

function preprocess(x, successor)

        //  total number of nodes in the graph
        n = successor.size

        //  initialize a dp table to store the 2^j th successor for each node

        dp = array[n][MaxLog]

        //  the 2^0 = 1st successor of the node is successor[i]
        for i = 1 to n
                dp[i][0] = successor[i]


        for j = 1 to MaxLog
                for i = 1 to n
                        //  calculate the value of dp[i][j] using dp[i][j - 1]
                        dp[i][j] = dp[dp[i][j - 1]][j - 1]

        return dp


/*
        Function to find the kth successor of node x in log(k) time.
*/

function succ(x, k, successor)

        //  if not preprocessed yet, call preprocess
        if not preprocessed
                dp = preprocess(x, successor)
                preprocessed = 1


        //  initialize the current node to x
        current_node = x
```

```
for i from 0 to log k
        if the ith bit of k is set
                // update the current node using dp table if the ith bit of K is set
                current_node = dp[current_node][i]

//   return current_node
return current_node
```

Time Complexity: O(logK) per query, where ‘K’ is the number of successors we are interested in. Also it has a preprocessing cost of **O(NlogK),** where N is the number of nodes in the graph. Preprocessing takes O(NlogK) time because we are populating the dp table of size (N*logK) and each update takes O(1) time. Also for each query we are running a single loop of size O(LogK).

Space Complexity: O(NLogK), where ‘N’ is the number of nodes in the given directed graph and ‘K’ is the number of successors we are interested in. The space used by the dp array is O(N*LogK) and we are not using any additional arrays anywhere else.