Date...................

As the tree is going to be half in every step it means the height of the tree is '$\log_2 n$'. Means, nc operations are performing '$\log_2 n$' times.
Means, our time complexity is :-

$O(nc \log_2 n)$

ignore this constant

so, ultimately the time complexity is :-

$\boxed{O(n \log_2 n)}$

Bit Manipulation

—— x —— x —— x ——

(Bit Manipulation Algorithms) (hackerearth)

Bits are actually very important from competitive point of view. In CP (competitive programming), if our submission differs by just few microseconds, then it affects your rank as well.
If we use BITWISE concepts then our submission will be fast.

working on bytes or data types comprising of bytes like int, float, double or even data structures which stores large amount of bytes is normal for a programmer. In some cases, a program

Date.....................

-mer needs to go beyond this - that is to say that in a deeper level where the importance of bits is realized.

Operations with bits are used in Data compression (data is compressed by conve-rting it from one representation to another, to reduce the space).
Exclusive-Or Encryption (an algorithm to encrypt the data for safety issues). In order to encode, decode or compress files we have to extract the data at bit level.
Bitwise Operations are faster & closer to the system & sometimes optimize the program to a good level.
We all know that 1 byte comprises of 8 bits & any integer or character can be represented using bits in computers, which we call its binary form (contain only 1 or 0) or in its base 2 form.
Example :-

(1) Binary Form of 14 is :-

$$14 \rightarrow (1110)_2$$

means,

$$\Rightarrow 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$$
$$\Rightarrow 8 + 4 + 2 + 0$$
$$\Rightarrow 14.$$

Date.................

ASCII codes

(2) Binary form of 20 is :-

$$20 \rightarrow (10100)_2$$

means,

$$\Rightarrow 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0$$

$$\Rightarrow 16 + 0 + 4 + 0 + 0$$

$$\Rightarrow 20.$$

For characters we use ASCII representation which are in the form of integers which again can be represented using bits as explained above.

Decimal (ASCII)	Hexa Decimal	character	Binary Equivalent
65	41	A	01000001
66	42	B	01000010
67	43	C	01000011
68	44	D	01000100
69	45	E	01000101
70	46	F	01000110
71	47	G	01000111
72	48	H	01001000
73	49	I	01001001
74	4A	J	01001010
75	4B	K	01001011
76	4C	L	01001100
77	4D	M	01001101
78	4E	N	01001110

Date.....................

Decimal (ASCII)	Hexa Decimal	Character	Binary Equivalent
79	4F	O	01001111
80	50	P	01010000
81	51	Q	01010001
82	52	R	01010010
83	53	S	01010011
84	54	T	01010100
85	55	U	01010101
86	56	V	01010110
87	57	W	01010111
88	58	X	01011000
89	59	Y	01011001
90	5A	Z	01011010
97	61	a	0110 0001
98	62	b	0110 0010
99	63	c	0110 0011
100	64	d	0110 0100
101	65	e	0110 0101
102	66	f	0110 0110
103	67	g	0110 0111
104	68	h	0110 1000
105	69	i	0110 1001
106	6A	j	0110 1010
107	6B	k	0110 1011
108	6C	l	0110 1100
109	6D	m	0110 1101
110	6E	n	0110 1110
111	6F	o	0110 1111

Date.....................

Decimal (ASCII)	Hexa Decimal	Character	Binary Equivalent
112	70	p	01110000
113	71	q	01110001
114	72	r	01110010
115	73	s	01110011
116	74	t	01110100
117	75	u	01110101
118	76	v	01110110
119	77	w	01110111
120	78	x	01111000
121	79	y	01111001
122	7A	z	01111010

Bitwise operators :- There are different bitwise operations used in the Bit Manipulation. These Bit Operations operate on the individual bits of the bit patterns. Bit Operations are fast & can be used in optimizing time complexity. Some common bit operations are :-

i) NOT (~) = Bitwise NOT is an unary operator that flips the bits of the number, i.e. if the ith bit is 0, it will change it to 1 and vice versa. Bitwise NOT is nothing but simply the one's complement of a number. For example :- $5 \rightarrow (101)_2$

$\sim 5 \rightarrow \sim(101)_2 \rightarrow (010)_2 = 2$.

Date..................

ii) AND (&) = Bitwise AND is a binary operator that operates on two equal length bit patterns. If both bits in the compared position of the bit patterns are 1, the bit in the resulting bit pattern is 1, otherwise 0.

For example, $5 \rightarrow (101)_2$
$3 \rightarrow (011)_2$

5 & 3 = 101
011
001 ⇒ $(001)_2$ ⇒ 1

iii) OR (|) = Bitwise OR is also binary operator that operates on two equal length bit patterns, similar to Bitwise AND. If both bits in the compared position of the bit patterns are 0, the bit in the resulting bit pattern is 0, otherwise 1.

For example, $5 \rightarrow (101)_2$
$3 \rightarrow (011)_2$

5 | 3 = 101
011
111 ⇒ $(111)_2$ ⇒ 7

iv) XOR (^) = Bitwise XOR also takes two equal length bit patterns. If both bits in the compared position of the bit patterns are 0 or 1, the bit in the resulting bit pattern is 1, otherwise 0.

Date.....................

For example : 5, $(101)_2$
3, $(011)_2$

5 ^ 3 = 101
011
———
110 => $(110)_2$ => 6

v) Left shift (<<) = Left shift operator is a binary operator which shift the some numbers of bits, in the bit pattern given, to the left & append 0 at the end. Left shift is equivalent to multiplying the bit pattern with 2^k (if we are shifting k bits).
For example,
1 << 1 → it means shift 1 by 1 bit.

Binary of 1 is,
0001 → left shift this by 1.
8 4 2 1
Resulting binary equivalent ← 0010
→ * left shift 1 by 1 bit, becomes 2.

1 << 2 → it means shift 1 by 2 bits.
Binary equivalent of 1 is,
0001 → left shift this by 2 bits
8 4 2 1
Resulting Binary equivalent ← 0100
→ * left shift 1 by 2 bits, becomes 4.

Date......................

1 << 3 → it means shift 1 by 3 bits

Binary equivalent of 1 is,

0 0 0 1 → left shift this by 3

8 4 2 1

Resulting ← 1 0 0 0
binary equivalent

→ * left shift 1 by 3 bits becomes 8.

vi) Right Shift (>>) : Right Shift operator is a binary operator which shift the some number of bits, in the given bit pattern, to the right & append 1 at the end, Right shift is equivalent to dividing the bit pattern with 2^k, (if we are shifting k bits).

For example,

4 >> 1 → it means shift 4 by 1 bit.

Binary equivalent of 4 is,

0 1 0 0 → right shift this by 1 bit

8 4 2 1

Resulting ← 0 0 1 0
binary equivalent

→ * Right shift 4 by 1 bit, becomes 2.

Date..................

6 >> 1 → right shift 6 by 1 bit
Binary equivalent of 6 is,
0 1 1 0
8 4 2 1
0 0 1 1

* Right shift 6 by 1 bit, becomes 3.

x	y	x&y	x|y	x^y	~x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bitwise operators are good for saving space & sometimes to cleverly remove dependenci-es.

(1) How to check if a given number is a power of 2?
Consider a number N and we need to find if N is a power of 2 or not. Simple solution to this problem is to repeated divide N by 2 if N is even. If we end up with a 1 then N is power of 2, otherwise not. There is a special case also. If N = 0 then it is not a power of 2.

Date.....................

The same problem can be solved using Bit Manipulation.

				128	64	32	16	8	4	2	1
2^0	→	1	→	0	0	0	0	0	0	0	1
2^1	→	2	→	0	0	0	0	0	0	1	0
2^2	→	4	→	0	0	0	0	0	1	0	0
2^3	→	8	→	0	0	0	0	1	0	0	0
2^4	→	16	→	0	0	0	1	0	0	0	0
2^5	→	32	→	0	0	1	0	0	0	0	0
2^6	→	64	→	0	1	0	0	0	0	0	0
2^7	→	128	→	1	0	0	0	0	0	0	0

let suppose if we want 2^4, the simple method is that shift 1 by 4 times, i.e., $1 << 4$.

If we want 2^7, then left shift 1 by 7 times, i.e., $1 << 7$.

$$\begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & 128 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & \\ 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & \end{array}$$

If we want 2^x, then left shift 1 by x times, i.e., $1 << x$.

$$\begin{array}{cc} 1 & \underbrace{0\ 0\ 0\ 0\ 0 - \cdots - 0}_{x \text{ time } 0\text{'s}} \rightarrow 2^0 \\ \downarrow & \downarrow \\ 2^x & 2^{x-1} \end{array}$$

Date.................

For example, num = 32, & we have to check than num is the power of 2 or not.

Bit manipulation way of doing this is simply calculate the binary equivalent of given num & also find the binary equivalent of num - 1.

Means, find the binary equivalent of 32 & 31 both.

Binary Equivalent of 32 is,

32	16	8	4	2	1
1	0	0	0	0	0

Binary Equivalent of 31 is,

32	16	8	4	2	1
0	1	1	1	1	1

Now perform the AND(&) operation on both binary numbers,

perform AND operation
```
  1 0 0 0 0 0
  0 1 1 1 1 1
  -----------
  0 0 0 0 0 0  → & operation gives all 0's.
```

In simple terms, if AND (&) operation on num & num - 1 gives all 0's. It means num is the power of 2.

Date.....................

Ex 2, num = 64.
num - 1 = 63.

Binary Equivalent of 64,
1000000

Binary Equivalent of 63,
0111111

Perform AND (&) operation,

```
 1000000
 0111111
 -------
 0000000  → & operation
            gives all zeroes.
```

It means 64 is the power of 2.

Ex 3, num = 20
num - 1 = 19.

```
               16 8 4 2 1
Binary of 20,  1  0 1 0 0
Binary of 19,  1  0 0 1 1
```

```
 10100
 10011
 -----
 10000  → Here, there is 1
          present in the
          & operation
```

It means 20 is not the power of 2.

Date.....................

(2) check if a given number is even & odd using bit manipulation.

Consider a number N & find even or odd, Simple solution to this problem is do (N % 2), if N % 2 == 0 then no. is even else if N % 2 == 1, then no. is odd.

The same problem can be solved using bit manipulation.

For example,

2 → 0 0 1 0
4 → 0 1 0 0
6 → 0 1 1 0
8 → 1 0 0 0
10 → 1 0 1 0

3 → 0 0 1 1
5 → 0 1 0 1
7 → 0 1 1 1
9 → 1 0 0 1

if we find out the pattern, in this binary numbers, in the right most bit even numbers has 0 & odd numbers has 1.

Let suppose, given num is 8 & we find that 8 is even or odd.

Simply find binary equivalent of 8, i.e, 1000 & perform AND (&) operati-on with 1's binary equivalent, i.e; 0001

Date......................

```
  1 0 0 0
  0 0 0 1
  -------
  0 0 0 0
```

→ if & operation gives all 0's, means given num is even else if & operation gives 1 at any place, means given num is odd.

* code (check power of 2) :-

```
#include<iostream>
using namespace std;
int main()
{
    int num;
    cout << "\n enter number to find
              its power of 2 or not:";
    cin >> num;
    int val = (num & (num -1));
    if( val == 0)
    {
      cout << "\n Yes";
    }
    else
    {
      cout << "\n NO";
    }
    return 0;
}
```

Date..................

* code (check even & odd):-

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "\n enter number: ");
    cin >> num;
    if(num & 1 ==1)
    {
        cout << "\n Odd.";
    }
    else
    {
        cout << "\n Even";
    }
}
```

Date..................

(3) Swap two numbers using bitwise operator.
By swapping, we exchange the value at two different locations in memory.
There are many ways to swap numbers in C++.
To perform swapping through bitwise operators, we have to use XOR operator.

Suppose, we have a number 23 stored in a variable x.
Binary of 23 is,
00010111
Other number is 34 stored in a variable y,
Binary of 34 is,
00100010

Step 1, Do bitwise XOR operation for x and y & store the result in x.

00010111
00100010
XOR operation in x & y ← 00110101 → replace the value of x with this value.

Date..................

Now,
x = 00110101
y = 00100010

Step 2, Do bitwise XOR operation for y and new value of x & store the result in y.

```
  00100010
  00110101
  --------
  00010111
```

XOR operation in y & new value of x. ← 00010111 → replace the value of y with this value

Now,
x = 00110101
y = 00010111

Step 3, Do bitwise XOR operation again for x and y & store the result in x.

```
  00110101
  00010111
  --------
  00100010
```

00100010 → replace this value of x with this value

Date.....................

Now,

	128	64	32	16	8	4	2	1
X =	0	0	1	0	0	0	1	0
Y =	0	0	0	1	0	1	1	1

Decimal equivalent of this is, 34

Decimal equivalent of this is, 23

After step 3, X = 34, Y = 23. } numbers are swapped

* code (swapping) :-

```
#include <iostream>
using namespace std;
int main()
{
    int x = 23, y = 34;
    cout << "before swapping x = " << x
              << " and y = " << y;

    x = x ^ y;
    y = x ^ y;     } swapping using bitwise
    x = x ^ y;       operator.

    cout << "after swapping x = " << x
              << " and y = " << y;

    return 0;
}
```