```
int power(int x, unsigned int y)
```

{ `if (y == 0)`
`return 1;`
else return(x* `power(x,y-1));`

} ===>O(n)

$z = x^y = x^{y/2} * x^{y/2}$, `if y is even`

```
int power(int x, unsigned int y)
{
    if (y == 0)
        return 1;
    else if (y % 2 == 0)
        return power(x, y / 2) * power(x, y / 2);
    else
        return x * power(x, y / 2) * power(x, y / 2);
}
```

T(n)=2T(n/2)+ O(1)....master theorem says---O(n)

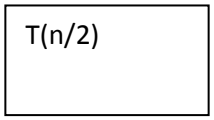
Better version:

logarithmic Paradigm: Divide and conquer.
Above function can be optimized to O(logn) by calculating power(x, y/2) only once and storing it.

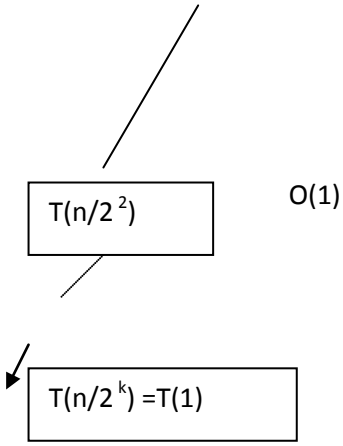
```
/* Function to calculate x raised to the power y in
O(logn)*/

int power(int x, unsigned int y)
{
    int temp;
    if( y == 0)
        return 1;
    temp = power(x, y / 2);
    if (y % 2 == 0)
        return temp * temp;
    else
        return x * temp * temp;
}
```

T(n)= T(n/2) +O(1)----O(logn) (master theorem)



O(1)



$n/2^k=1 \Rightarrow k=\log n$

$T(n)= O(1) + O(1) + \ldots$ upto k terms $= O(k)= O(\log n)$