Recursion Runtime Tables

```
unsigned int factorial(unsigned int n)
{
    if (n <= 1)
    {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Step	Statement	Cost	Time
1	`n <= 1`	c_1	1
2	`return 1;`	c_2	0
3	`return n * factorial(n - 1);`	c_3	$1 + 2(n - 1) = 2n - 1$

$T(n) = c_1 + c_3(2n - 1)$

$= c_3(2n) + (c_1 - c_3)$

$= 2n$

Big-O = $O(n)$

The second return statement has a time of $1 + 2(n - 1)$.

1 for the return statement itself.

$2(n - 1)$ because the number of recursive calls is $n - 1$.

There are 2 steps in the function.

Therefore, the number of calls is multiplied by 2.

Subtotal: $1 + 2(n - 1) = 1 + 2n - 2 = 2n - 1$

Total: $1 + 2n - 1 = 2n$

```
unsigned int digitCount(unsigned int n)
{
    if (n < 10)
    {
        return 1;
    }
    return 1 + digitCount(n / 10);
}
```

Step	Statement	Cost	Time
1	n < 10	c_1	1
2	return 1;	c_2	0
3	return 1 + digitCount(n / 10);	c_3	$1 + 2\lfloor \log_{10}(n) \rfloor$

$T(n) = c_1 + c_3 + c_3(2\lfloor \log_{10}(n) \rfloor)$

$= c_3(2\lfloor \log_{10}(n) \rfloor) + (c_1 + c_3)$

$= 2\lfloor \log_{10}(n) \rfloor + 2$

Big-O = $O(\log(n))$

n is the value of the parameter n

The second return statement has a time of $1 + 2\lfloor \log_{10}(n) \rfloor$

1 for the return statement itself.

$2\lfloor \log_{10}(n) \rfloor$ because the number of recursive calls is $\lfloor \log_{10}(n) \rfloor$.

There are 2 steps in the function.

Therefore, the number of calls is multiplied by 2.

Subtotal: $2\lfloor \log_{10}(n) \rfloor + 1$

Total: $2\lfloor \log_{10}(n) \rfloor + 2$

```
template <class T>
class Node
{
    public:
        T data;
        Node<T>* next;
        Node(const T& value) : data(value), next(nullptr) {}
};
bool contains(Node<int>* node, int target)
{
    if (node == nullptr)
    {
        return false;
    }
    if (node->data == target)
    {
        return true;
    }
    return contains(node->next, target);
}
```

Step	Statement	Cost	Time
1	`node == nullptr`	c_1	1
2	`return false;`	c_2	0
3	`node->data == target`	c_3	1
4	`return true;`	c_4	0
5	`return contains(node->next, target);`	c_5	2n

$T(n) = c_1 + c_3 + c_5(2n)$

$= 2c_5(n) + (c_1 + c_3)$

$= 2n + 2$

Big-O = O(n)

n = The number of nodes in the linked list pointed to by head.

The third return statement has a time of 2n. Since there is no operation being performed on the return statement, only the recursive calls get counted. 2n because the number of recursive calls is n and there are 2 steps in the function. The number of calls is multiplied by 2.

Subtotal: 2n

Total: 2n + 2

```cpp
void printBinaries(unsigned int n, const string& b = "")
{
    if (b.length() == n)
    {
        cout << b << endl;
        return;
    }
    printBinaries(n, b + "0");
    printBinaries(n, b + "1");
}
```

Step	Statement	Cost	Time
1	`b.length() == n`	c_1	1
2	`cout << b << endl;`	c_2	0
3	`return;`	c_3	0
4	`printBinaries(n, b + "0");`	c_4	$2^n + 2^{n-1} - 1$
5	`printBinaries(n, b + "1");`	c_5	$2^n + 2^{n-1} - 1$

$$T(n) = c_1 + (c_4 + c_5)(2^n + 2^{n-1} - 1)$$

$$= 2(2^n + 2^{n-1} - 1) + 1$$

$$= 2^{n+1} + 2^n - 2 + 1$$

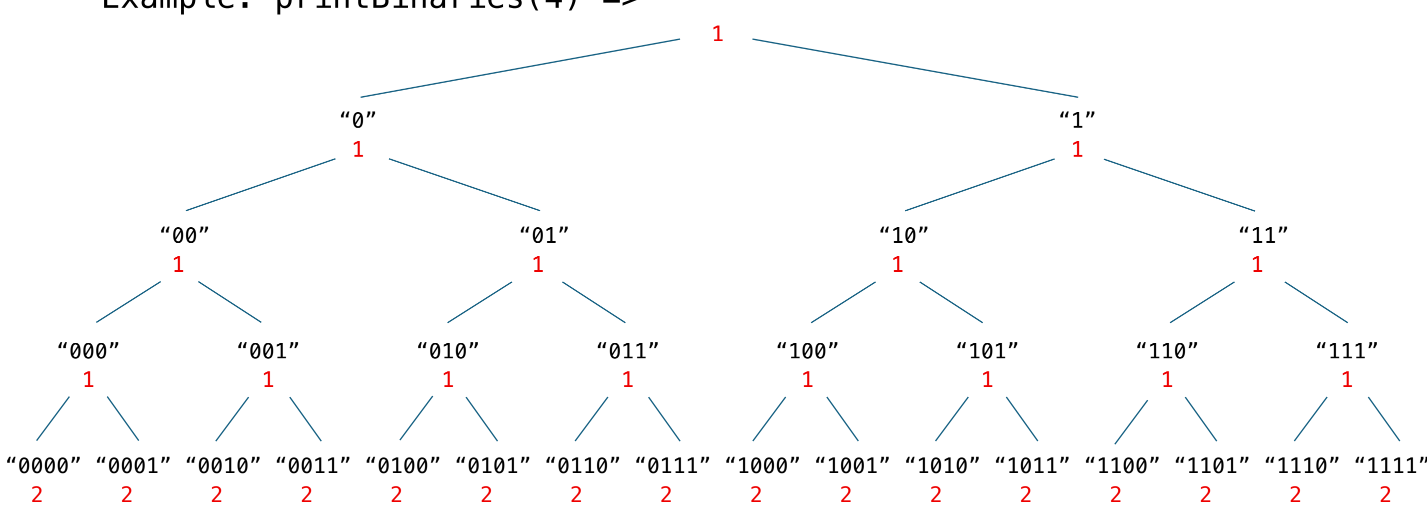
$$= (2^n)(2+1) - (2 - 1)$$

$$= 3(2^n) - 1$$

Big-O = $O(2^n)$

n = n (characters) in the string.

Example: printBinaries(4) => “”



Since this is a void function, the return statement is ignored. Each recursive call has a time of $2^n + 2^{n-1} - 1$. Whenever the function is called, the function calls itself twice and does so n times. The sum of all the recursive calls is $2^n - 1$.

$$\sum_{i=1}^{n} 2^{i-1} = 2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$$

Since one step gets executed per call, $2^n - 1$ remains. However, when the length of the string reaches n, there are 2 steps for each function call and 2^{n-1} calls. For these calls, 1 can be added for each for a total of 2^{n-1}. The time now is $2^n + 2^{n-1} - 1$.

Subtotal: $2^n + 2^{n-1} - 1$

Total: $2(2^n + 2^{n-1} - 1) + 1 = 3(2^n) - 1$