O(log n) Time Complexity

Time complexity is a way of measuring the performance of algorithms by estimating how much time they take to run for a given input size. It is important to analyze the time complexity of algorithms because it helps us to compare different solutions, optimize our code, and understand the trade-offs and limitations of our algorithms.

One of the most common ways of expressing the time complexity of an algorithm is using the Big O notation. The Big O notation describes the worst-case scenario of an algorithm's growth rate, or how fast its running time increases as the input size increases. For example, an algorithm that has a time complexity of **O(n)** means that its running time is proportional to the input size **n**, or that it takes n steps to complete. An algorithm that has a time complexity of **$O(n^2)$** means that its running time is proportional to the square of the input size **n**, or that it takes **n^2** steps to complete.

There are many different time complexities that can be expressed using the Big O notation, such as **O(1), O(log n), O(n log n), $O(n^2)$, O(2^n)**, etc. Each of these time complexities represents a different class of algorithms that have different performance characteristics and implications. In this article, we will focus on one of the most common and useful time complexities: logarithmic time complexity, or **O(log n)**.

Logarithms and Exponents

Logarithmic time complexity means that the algorithm's running time grows proportionally to the logarithm of the input size. But what is a logarithm and how does it relate to the input size?

A logarithm is a mathematical operation that is the inverse of exponentiation. Exponentiation is when we raise a number to a power, such as $2^3 = 8$. Logarithm is when we find what power we need to raise a number to get another number, such as **log_2(8) = 3**. In other words, logarithm is asking the question: how many times do we need to multiply a number by itself to get another number?

Logarithms can have different bases, which are the numbers that we multiply by themselves. For example, **log_2(n)** means how many times do we need to multiply 2 by itself to get n, while **log_10(n)** means how many times do we need to multiply 10 by itself to get n. The base of a logarithm affects its value, but not its growth rate. We can convert between logarithms with different bases using the change-of-base formula: **log_b(x) = log_a(x) / log_a(b). For example, log_2(8) = log_10(8) / log_10(2) = 3 / 0.301 = 9.966**.

Logarithms can be used to measure how many times an input can be divided by a constant factor. For example, **log_2(n)** means how many times do we need to divide n by 2 until we get 1, while **log_10(n)** means how many times do we need to divide n by 10 until we get 1. This property of logarithms is very useful for analyzing algorithms that work by repeatedly dividing the input size by a constant factor until they reach a base case.

We can calculate the logarithm of a number using a calculator or a programming language. For example, in Python, we can use the **math.log** function with an optional base argument:

```python
import math
print(math.log(8)) # default base is e (natural logarithm)
print(math.log(8, 2)) # base 2 (binary logarithm)
print(math.log(8, 10)) # base 10 (common logarithm)
```

The output is:

```
2.0794415416798357
3.0
0.9030899869919435
```

O(log n) Algorithms

Now that we understand what logarithms are and how they work, let’s see some examples of algorithms that have logarithmic time complexity.

The general idea of how **O(log n)** algorithms work is that they repeatedly divide the input size by a constant factor until they reach a base case. This means that in each step, they reduce the problem size significantly, which makes them faster than linear or quadratic algorithms for large inputs.

Here are some examples of **O(log n)** algorithms with pseudocode and code snippets in Python:

Binary Search

Binary search is an algorithm that finds an element in a sorted array by comparing it with the middle element and discarding half of the array in each iteration. It works as follows:

- If the array is empty, return -1 (not found).
- Find the middle index and element of the array.
- If the element is equal to the target value, return the index (found).
- If the element is less than the target value, discard the left half of the array and repeat the process with the right half.
- If the element is greater than the target value, discard the right half of the array and repeat the process with the left half.

The pseudocode for binary search is:

```
function binary_search(array, target):
  low = 0
  high = length of array - 1
  while low <= high:
    mid = (low + high) / 2
    element = array[mid]
    if element == target:
      return mid
    elif element < target:
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

The Python code for binary search is:

```
def binary_search(array, target):
  low = 0
  high = len(array) - 1
  while low <= high:
    mid = (low + high) // 2 # integer division
    element = array[mid]
    if element == target:
      return mid
    elif element < target:
      low = mid + 1
    else:
      high = mid - 1
  return -1

# example usage
array = [1, 3, 5, 7, 9]
target = 5
index = binary_search(array, target)
print(index) # output: 2
```

The time complexity of binary search is **O(log n)** because in each iteration, it discards half of the array, which means that it divides the input size by 2. Therefore, the number of iterations is proportional to **log_2(n)**, where n is the length of the array.

Finding the Height of a Binary Tree

A binary tree is a data structure that consists of nodes that have at most two children: a left child and a right child. The height of a binary tree is the maximum depth of any node in the tree, or the number of edges from the root node to the farthest leaf node.

Finding the height of a binary tree is an algorithm that calculates the maximum depth of a binary tree by recursively finding the height of its left and right subtrees and adding one. It works as follows:

- If the tree is empty, return -1 (base case).
- Find the height of the left subtree by calling the function recursively on the left child.
- Find the height of the right subtree by calling the function recursively on the right child.
- Return the maximum of the left and right heights plus one.

The pseudocode for finding the height of a binary tree is:

```
function height(tree):
  if tree is empty:
    return -1
  left_height = height(tree.left)
  right_height = height(tree.right)
  return max(left_height, right_height) + 1
```

The Python code for finding the height of a binary tree is:

```
# define a class for binary tree nodes
class TreeNode:
  def __init__(self, val):
    self.val = val # node value
    self.left = None # left child
    self.right = None # right child

# define a function for finding the height of a binary tree
def height(tree):
  if tree is None: # base case
    return -1
  left_height = height(tree.left) # recursive call on left child
  right_height = height(tree.right) # recursive call on right child
  return max(left_height, right_height) + 1 # return maximum plus one

# example usage
tree = TreeNode(1) # create a root node with value 1
tree.left = TreeNode(2) # create a left child with value 2
tree.right = TreeNode(3) # create a right child with value 3
tree.left.left = TreeNode(4) # create a left grandchild with value 4
tree_height = height(tree) # find the height of the tree
print(tree_height) # output: 2
```

The time complexity of finding the height of a binary tree is **O(log n)** if the tree is balanced, meaning that its left and right subtrees have roughly equal heights. In this case, in each recursive call, we reduce the problem size by half, which means that we divide the input size by 2. Therefore, the number of recursive calls is proportional to **log_2(n)**, where n is the number of nodes in the tree.

However, if the tree is unbalanced, meaning that its left and right subtrees have very different heights, then the time complexity can be as bad as **O(n)**, where n is still the number of nodes in the tree. In this case, in each recursive call, we reduce the problem size by only one node, which means that we do not divide the input size by any constant factor. Therefore, the number of recursive calls is proportional to n.

Binary Exponentiation

Binary exponentiation is an algorithm that computes a power of a number by repeatedly squaring it and multiplying it by itself if the exponent is odd. It works as follows:

- If the exponent is zero, return 1 (base case).
- If the exponent is even, return the square of the result of raising the number to half of the exponent (recurrence relation).
- If the exponent is odd, return the number multiplied by the result of raising the number to one less than the exponent (recurrence relation).

The pseudocode for binary exponentiation is:

```
function power(base, exponent):
  if exponent == 0:
    return 1
  elif exponent % 2 == 0: # even exponent
    return power(base, exponent / 2) * power(base, exponent / 2)
  else: # odd exponent
    return base * power(base, exponent - 1)
```

The Python code for binary exponentiation is:

```
def power(base, exponent):
  if exponent == 0: # base case
    return 1
  elif exponent % 2 == 0: # even exponent
    return power(base, exponent // 2) ** 2 # integer division and squaring
  else: # odd exponent
    return base * power(base, exponent - 1) # multiplication and subtraction

# example usage
base = 2
exponent = 10
result = power(base, exponent)
print(result) # output: 1024
```

The time complexity of binary exponentiation is **O(log n)**, where n is the exponent. This is because in each recursive call, we either divide the exponent by 2 or subtract it by 1, which means that we reduce the problem size by a constant factor. Therefore, the number of recursive calls is proportional to **log_2(n)**, where n is the exponent.

Advantages and Disadvantages of O(log n) Algorithms

O(log n) algorithms have some advantages and disadvantages that we should be aware of when using them. Here are some of them:

Advantages

- They are faster than linear or quadratic algorithms for large inputs, as they reduce the problem size significantly in each step.
- They are often simple and elegant, as they use divide-and-conquer or recursion techniques.

Disadvantages

- They may require extra space or memory for storing intermediate results or recursive calls.

- They may not be optimal for small inputs, as they may have a high constant factor or overhead.

Conclusion

We have learned about logarithmic time complexity, or **O(log n)**, which indicates that an algorithm's running time grows proportionally to the logarithm of the input size. We have seen what logarithms are and how they measure how many times an input can be divided by a constant factor. We have also seen some examples of **O(log n)** algorithms, such as binary search, finding the height of a binary tree, and binary exponentiation. We have also discussed some advantages and disadvantages of **O(log n)** algorithms.

Reference:

Kuldeep Singh 2023, *O (log n) time complexity: A Comprehensive Analysis*, accessed 14 January 2024, <https://kd-singh.medium.com/o-log-n-time-complexity-a-comprehensive-analysis-703a9fb42565>