[CS203] Design and Analysis of Algorithms

Course Instructor: Dr. Dibyendu Roy
Scribed by: Prakhar Jain (202251099)

Autumn 2023-24
Lecture (Week 3)

We use big-O notation to describe how a function grows, capturing the growth without relying on specific details like processor speed and without worrying about small inputs.

For functions $f(x)$ and $g(x)$, we say "f(x) is $O(g(x))$" (pronounced "f(x) is big-oh of g(x)") if there are positive constants C and k such that

$$|f(x)| \leq C|g(x)| \text{ for all } x > k.$$

This notation gives us a way to describe a function's growth.
Example: The function $f(x) = 2x^3 + 10x$ is $O(x^3)$.
Proof: To satisfy the definition of big-O, we find values for C and k.
Let $C = 12$ and $k = 2$. Then for $x > k$,

$$|2x^3 + 10x| = 2x^3 + 10x < 2x^3 + 10x^3 = |12x^3|.$$

1 Time Complexity Analysis

In computer science, we use various notations to analyze the time complexity of algorithms, including Big O notation ($O(g(n))$), Omega notation ($\Omega(g(n))$), and Theta notation ($\Theta(g(n))$).

1.1 Big O Notation ($O(g(n))$)

Big O notation provides an upper bound on the growth rate of a function. It represents the worst-case time complexity of an algorithm. For example, if we have a function $f(n)$ and we say $f(n) = O(g(n))$, it means that $f(n)$ will not grow faster than some constant multiple of $g(n)$ for large values of n.

1.2 Omega Notation ($\Omega(g(n))$)

Omega notation provides a lower bound on the growth rate of a function. It represents the best-case time complexity of an algorithm. If we have a function $f(n)$ and we say $f(n) = \Omega(g(n))$, it means that $f(n)$ will not grow slower than some constant multiple of $g(n)$ for large values of n.

1.3 Theta Notation ($\Theta(g(n))$)

Theta notation provides both upper and lower bounds on the growth rate of a function. It represents the average-case time complexity of an algorithm. If we have a function $f(n)$ and we say $f(n) = \Theta(g(n))$, it means that $f(n)$ grows at the same rate as $g(n)$ within constant factors for large values of n.

2 Maximum Subarray Problem

In the context of algorithm analysis, the Maximum Subarray Problem involves finding a subarray within an array that has the maximum sum. This problem can be solved efficiently using the divide-and-conquer approach.

2.1 Divide-and-Conquer Algorithm

The divide-and-conquer algorithm for solving the Maximum Subarray Problem can be summarized as follows:

1. Divide the given array into two halves, creating left and right subarrays.
2. Recursively find the maximum subarray in the left and right subarrays.
3. Find the maximum subarray that crosses the midpoint of the array.
4. Return the subarray with the maximum sum among the three: left, right, and cross-midpoint subarrays.

Algorithm 1 Find Max Crossing Subarray

procedure FINDMAXCROSSINGSUBARRAY(A, low, high)
  $left\_sum \leftarrow -\infty$ ▷ Initialize maximum left subarray sum
  $right\_sum \leftarrow -\infty$ ▷ Initialize maximum right subarray sum
  $mid \leftarrow (\text{low} + \text{high})//2$ ▷ Find middle index
  Find maximum subarray sum on the left side:
  $sum \leftarrow 0$ ▷ Initialize sum of current subarray
  for $i \leftarrow$ mid **downto** low **do**
    $sum \leftarrow sum + A[i]$
    if $sum > left\_sum$ **then**
      $left\_sum \leftarrow sum$
  Find maximum subarray sum on the right side:
  $sum \leftarrow 0$ ▷ Initialize sum of current subarray
  for $i \leftarrow \text{mid} + 1$ **to** high **do**
    $sum \leftarrow sum + A[i]$
    if $sum > right\_sum$ **then**
      $right\_sum \leftarrow sum$
  The maximum subarray that crosses the middle:
  $left\_idx \leftarrow$ index of the left boundary
  $right\_idx \leftarrow$ index of the right boundary
  The maximum subarray sum:
  $max\_sum \leftarrow left\_sum + right\_sum$
  Return the maximum subarray from $A[left\_idx]$ to $A[right\_idx]$ and max_sum

In Figure **??**, you can see how the array is divided into left and right subarrays during the divide-and-conquer process.

2.2 Time Complexity Analysis

To analyze the time complexity of the "Find Max Crossing Subarray" algorithm, we can use the recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n) + c$$

Where:

- $T(n)$ is the time complexity of the algorithm for an input of size n.
- $T\left(\frac{n}{2}\right)$ represents the time complexity for the left and right subproblems of size $\frac{n}{2}$.
- $\Theta(n)$ represents the time taken to compute the maximum subarray crossing the middle.
- c represents the time taken for other constant-time operations.

Now, let's solve this recurrence relation using the Master Theorem. The Master Theorem has the following form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

In our case, $a = 1$, $b = 2$, and $f(n) = \Theta(n) + c$.
Now, we need to compare $f(n)$ with $n^{\log_b(a)}$, where $a = 1$ and $b = 2$:

$$n^{\log_2(1)} = n^0 = 1$$

Here's how we compare:

1. If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$. In our case, this condition doesn't apply because $f(n)$ is not smaller than n^0.
2. If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \log n)$. In our case, this condition doesn't apply because $f(n)$ is not the same as n^0.
3. If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq kf(n)$ for some $k < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

In our case, $f(n)$ is indeed larger than n^0 ($f(n) = \Theta(n)$), and the regularity condition holds because:

$$af\left(\frac{n}{b}\right) = 1 \cdot \Theta\left(\frac{n}{2}\right) = \Theta\left(\frac{n}{2}\right)$$
$$kf(n) = k \cdot \Theta(n) = \Theta(n)$$

Since $k < 1$, the regularity condition is satisfied.
Therefore, by applying Case 3 of the Master Theorem, we can conclude that the time complexity of the algorithm is:

$$T(n) = \Theta(f(n)) = \Theta(n)$$

So, the time complexity of the "Find Max Crossing Subarray" algorithm is $\Theta(n)$.