CS 2500: Algorithms

Lecture 8: Master Theorem and Program Correctness

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

September 12, 2024

Divide and Conquer Recurrences

- Divide and Conquer algorithms split a problem into smaller subproblems, solve them recursively, and combine the results.
- Their time complexity can often be described using recurrence relations.
- Example: Merge Sort's recurrence relation:

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

Simplified Master Theorem

The Simplified Master Theorem is used to solve recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

where:

- $a \geq 1$: Number of subproblems.
- $b > 1$: Factor by which the problem size decreases.
- n^d: Cost of combining the subproblem solutions.

Master Theorem - Cases

There are three cases in the Master Theorem, based on the comparison between a and b^d:

Case 1: If $a > b^d$:

$$T(n) = O(n^{\log_b a})$$

Case 2: If $a = b^d$:

$$T(n) = O(n^d \log n)$$

Case 3: If $a < b^d$:

$$T(n) = O(n^d)$$

Example 1: Merge Sort

Let's apply the Master Theorem to the Merge Sort recurrence:

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

- Here, $a = 2$, $b = 2$, and $d = 1$.
- Compute $b^d = 2^1 = 2$.
- Since $a = b^d$, we are in **Case 2**.
- Therefore, the solution is:

$$T(n) = O(n \log n)$$

Example 2: Binary Search

Binary Search recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

- Here, $a = 1$, $b = 2$, and $d = 0$.
- Compute $b^d = 2^0 = 1$.
- Since $a = b^d$, we are in **Case 2**.
- Therefore, the solution is:

$$T(n) = O(\log n)$$

Example 3: Strassen's Algorithm

Strassen's matrix multiplication recurrence:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

- Here, $a = 7$, $b = 2$, and $d = 2$.
- Compute $b^d = 2^2 = 4$.
- Since $a > b^d$, we are in **Case 1**.
- Therefore, the solution is:

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

Example 3: Quicksort

Recurrence:

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

- Here, $a = 2$, $b = 2$, and $d = 1$.
- Compute $b^d = 2^1 = 2$.
- Since $a = b^d$, we are in **Case 2** of the Master Theorem.
- Therefore, the solution is:

$$T(n) = O(n \log n)$$

Example 4: Karatsuba's Fast Multiplication Algorithm

Recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

- Here, $a = 3$, $b = 2$, and $d = 1$.
- Compute $b^d = 2^1 = 2$.
- Since $a > b^d$, we are in **Case 1** of the Master Theorem.
- Therefore, the solution is:

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

Generalized Master Theorem

- The Generalized Master Theorem is an extension of the standard Master Theorem for more complex recurrence relations.
- It is applicable when the non-recursive term $f(n)$ is more complicated than a simple polynomial $O(n^d)$.
- Recurrence form:

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

- Where $f(n)$ can be any asymptotic function, not just $O(n^d)$.

Recurrence Form

The Generalized Master Theorem solves recurrences of the form:

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

where:

- $a \geq 1$: Number of subproblems.
- $b > 1$: Factor by which the problem size is reduced.
- $f(n)$: The cost of the work outside the recursive calls.

The theorem handles more complex $f(n)$ functions, such as logarithmic or exponential terms.

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$

Case 1: If $f(n)$ grows polynomially slower than $n^{\log_b a}$:

$$f(n) = O(n^{\log_b a - \epsilon})$$

for some $\epsilon > 0$, then:

$$T(n) = O(n^{\log_b a})$$

Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$

Case 2: If $f(n) = \Theta(n^{\log_b a} \log^k n)$, for some $k \geq 0$, then:

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

Case 3: $f(n) = \Omega(n^{\log_b a+\epsilon})$

Case 3: If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$, and if:

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

for some constant $c < 1$, then:

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

Example

Question. Solving the Recurrence:

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$

using Generalized Master Theorem
Step 1: Identify Parameters

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$

Compare with the standard form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where:

- $a = 8$
- $b = 2$
- $f(n) = n^2$

Step 2: Calculate $\log_b a$

$$\log_b a = \log_2 8 = 3$$

Example

Step 3: Compare $f(n)$ with $n^{\log_b a}$

- $f(n) = n^2$
- $n^{\log_b a} = n^3$

Since $f(n) = O(n^{3-1})$, we are in **Case 1**.

Step 4: Apply Case 1 of the Generalized Master Theorem

$$T(n) = O(n^{\log_b a}) = O(n^3)$$

Final Solution: The time complexity is $O(n^3)$.

Conclusion and Key Insights

- The Master Theorem provides an efficient way to analyze divide-and-conquer recurrences, particularly for polynomial cost functions.
- The Generalized Master Theorem extends the standard version to handle more complex cases where the non-recursive part grows faster or slower than simple polynomials (e.g., logarithmic or exponential terms).
- Understanding the relationship between a, b, d, and $f(n)$ is crucial for selecting the correct case in either theorem.
- Both theorems are powerful tools, but always check the assumptions and the form of the recurrence before applying them.
- Practice solving different recurrences to gain a deeper understanding and strengthen your skills in applying these theorems.

Recursive GCD Algorithm

Algorithm RecursiveGCD(a, b)

```
if b = 0 then                              ▷ Base Case
    return a
else                                       ▷ Recursive Step
    return RecursiveGCD(b, a mod b)
end if
```

What We Want to Prove

- The Recursive GCD algorithm correctly computes the greatest common divisor (GCD) of two numbers a and b.
- We will use **mathematical induction** on the value of b to prove correctness.

Proof by Induction: Base Case $b = 0$

Base Case: $b = 0$

- When $b = 0$, the algorithm returns a.
- According to the definition of GCD:

$$\text{GCD}(a, 0) = a$$

- This is true because any number divides 0, and the greatest divisor of a and 0 is a.
- Therefore, when $b = 0$, the algorithm correctly returns a, satisfying the base case.

Proof of Correctness: Inductive Hypothesis

Inductive Hypothesis:

- Assume that for all values $b' < b_0$, the algorithm correctly computes the GCD of a and b', i.e.,

$$\text{RecursiveGCD}(a, b') = \text{GCD}(a, b') \quad \text{for any } b' < b_0.$$

- We will now prove that the algorithm works for $b = b_0$.

Proof by Induction: Inductive Step for $b = b_0$

Inductive Step: Prove the algorithm is correct for $b = b_0$

- The algorithm calls:

$$\text{RecursiveGCD}(a, b_0) = \text{RecursiveGCD}(b_0, a \mod b_0)$$

- By the Euclidean algorithm, we know:

$$\text{GCD}(a, b_0) = \text{GCD}(b_0, a \mod b_0)$$

- This is a property of the GCD: the GCD of two numbers doesn't change if the larger number is replaced by its remainder when divided by the smaller number.

Inductive Step: Continuation

- The second argument in the recursive call is $a \mod b_0$, which is smaller than b_0, i.e., $a \mod b_0 < b_0$.
- By our **inductive hypothesis**, we assume that the algorithm correctly computes the GCD for all values smaller than b_0.
- Therefore, RecursiveGCD$(b_0, a \mod b_0)$ correctly computes GCD$(b_0, a \mod b_0)$.
- Since GCD$(a, b_0) =$ GCD$(b_0, a \mod b_0)$, the recursive call returns the correct value for GCD(a, b_0).

Conclusion of the Proof

- By mathematical induction, we have shown that:
  - The base case ($b = 0$) works correctly.
  - The inductive step holds, as the recursive call solves a smaller instance of the problem, and the GCD is computed correctly.
- Therefore, the **Recursive GCD Algorithm is correct** for all values of a and b.

Recursive Algorithm for Computing a^n

Algorithm Power(a, n)

procedure POWER(a, n) ▷ a is a non-zero real number, n is a non-negative integer
    if $n = 0$ **then** ▷ Base Case
        return 1
    else
        return $a \times \text{Power}(a, n - 1)$
    end if
end procedure

This algorithm recursively computes a^n by reducing the exponent.

What We Want to Prove

- We want to prove that the recursive algorithm correctly computes a^n for any non-negative integer n.
- We will use **mathematical induction** on n to prove its correctness.

Proof by Induction: Base Case $n = 0$

Base Case: $n = 0$

- When $n = 0$, the algorithm returns 1.
- This is correct because $a^0 = 1$ for any non-zero real number a.
- Therefore, the base case holds: $\text{Power}(a, 0) = 1$.

Inductive Hypothesis

- Assume that the algorithm correctly computes a^k for some arbitrary non-negative integer k.
- That is, assume $\text{Power}(a, k) = a^k$.
- We will now prove that the algorithm correctly computes a^{k+1}.

Inductive Step:

- When $n = k + 1$, the algorithm computes:

$$\text{Power}(a, k + 1) = a \times \text{Power}(a, k)$$

- By the inductive hypothesis, we know that $\text{Power}(a, k) = a^k$.
- Therefore:

$$\text{Power}(a, k + 1) = a \times a^k = a^{k+1}$$

- Thus, the algorithm correctly computes a^{k+1} when $n = k + 1$.

Conclusion of the Proof

- By mathematical induction, we have shown that:
  - The base case ($n = 0$) holds, as the algorithm returns 1, which is correct.
  - The inductive step holds, as the algorithm correctly computes a^{k+1} from a^k.
- Therefore, the recursive algorithm correctly computes a^n for any non-negative integer n.

Recursive Linear Search

Algorithm search(a, i, j, x)

if $a[i] = x$ **then**
  return i ▷ Found the element at index i
else if $i = j$ **then**
  return -1 ▷ Reached the end of the search range without finding x
else
  return search(a, $i + 1$, j, x) ▷ Continue searching in the rest of the array
end if

Proof by Induction: Base Case

Base Case: $j - i = 1$

- The subarray consists of a single element at index i.
- The algorithm checks whether $a[i] = x$:
  - If $a[i] = x$, it returns i, which is correct.
  - If $a[i] \neq x$, it checks whether $i = j$, and returns -1, correctly indicating x is not found.
- Thus, the algorithm works correctly for subarrays of size 1.

Proof by Induction: Inductive Hypothesis

Inductive Hypothesis:

- Assume the algorithm works correctly for all subarrays of size k, i.e., when $j - i = k$.
- That is, for any subarray of size k, the algorithm returns the correct index if x is found, or -1 if x is not found.

Proof by Induction: Inductive Step

Inductive Step:

- Now consider a subarray of size $k+1$, i.e., when $j-i=k+1$.
- The algorithm checks whether $a[i]=x$:
  - If $a[i]=x$, it returns i, which is correct.
  - If $a[i]\neq x$, it makes a recursive call to `search(a, i+1, j, x)`.
- By the inductive hypothesis, the recursive call works correctly for the remaining subarray of size k.
- Therefore, if x is found, the recursive call returns the correct index; otherwise, it returns -1.

Conclusion of the Proof

- By mathematical induction, the recursive linear search algorithm works correctly for any subarray of size $j - i \geq 1$.
- Therefore, the algorithm correctly finds the element x or returns -1 if x is not found.