1.5em 0pt

[CS203] Design and Analysis of Algorithms

Course Instructor: Dr. Dibyendu Roy — Autumn 2023-24

Scribed by: prakhar jain (202251099) — Lecture (Week 4)

1 Matrix Multiplication

In this section, we will discuss the standard matrix multiplication algorithm where we compute the product matrix C from two input matrices A and B, denoted as $C = A \cdot B$. This algorithm has a time complexity of $O(n^3)$, where n is the size of the input matrices.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

$$C = A \cdot B$$

The time complexity of this algorithm is $O(n^3)$, which means that the number of arithmetic operations required to compute the product grows cubically with the size of the matrices.

2 Derivation of Time Complexity Recurrence

Given the recurrence relation for the matrix multiplication algorithm:

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(1)$$

We assume that $n = 2^k$, where k is a positive integer representing the recursion depth.

2.1 Applying the Master Theorem

To solve this recurrence relation, we will apply the Master Theorem. The recurrence follows the form:

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

Where in our case, $a = 8$, $b = 2$, and $f(n) = \Theta(1)$.

2.1.1 Step 1: Calculate $n^{\log_b(a)}$

$$n^{\log_b(a)} = 2^{\log_2(8)} = 2^3 = 8$$

2.1.2 Step 2: Compare $f(n)$ and $n^{\log_b(a)}$

Since $f(n) = \Theta(1)$, which is polynomially smaller than $n^{\log_b(a)}$, we are in **Case 1** of the Master Theorem.

2.1.3 Step 3: Solve for $T(n)$

3 Proof of Strassen's Algorithm Time Complexity

In Strassen's Algorithm, we calculate the product of two matrices A and B to obtain the result matrix C. We define intermediate matrices S_i and P_i as follows:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

We also calculate P_i as follows:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 \\
P_2 &= S_2 \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 \\
P_5 &= S_5 \cdot S_6 \\
P_6 &= S_7 \cdot S_8 \\
P_7 &= S_9 \cdot S_{10}
\end{aligned}
$$

Now, let's analyze the time complexity of Strassen's Algorithm.

3.1 Time Complexity Analysis

In Strassen's Algorithm, we recursively calculate the values of P_i and use them to compute the quadrants of the result matrix C. Each step of the algorithm divides the matrices into smaller submatrices and performs multiplications, additions, and subtractions.

The time complexity of Strassen's Algorithm can be calculated using the following recurrence relation:

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

3.1.1 Applying the Master Theorem

To solve this recurrence relation, we apply the Master Theorem. The recurrence follows the form:

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

Where in our case, $a = 7$, $b = 2$, and $f(n) = \Theta(n^2)$.

3.1.2 Step 1: Calculate $n^{\log_b(a)}$

$$n^{\log_b(a)} = n^{\log_2(7)}$$

3.1.3 Step 2: Compare $f(n)$ and $n^{\log_b(a)}$

Since $f(n) = \Theta(n^2)$, which is polynomially smaller than $n^{\log_b(a)}$, we are in **Case 1** of the Master Theorem.

3.1.4 Step 3: Solve for $T(n)$

In **Case 1**, the solution for $T(n)$ is:

$$T(n) = \Theta\left(n^{\log_b(a)} \log^k(n)\right)$$

Since $f(n) = \Theta(n^2)$, $k = 1$.
Thus, the time complexity of Strassen's Algorithm is:

$$T(n) = \Theta\left(n^{\log_2(7)} \log(n)\right)$$

This can be simplified to:

$$T(n) = \Theta(n^{\log_2(7)})$$

3.1.5 Time Complexity Interpretation

The time complexity of Strassen's Algorithm is $\Theta(n^{\log_2(7)})$, which is approximately $\Theta(n^{2.81})$. This complexity is lower than the standard matrix multiplication algorithm, which has a time complexity of $\Theta(n^3)$.

3.2 Conclusion

Strassen's Algorithm reduces the time complexity of matrix multiplication from $O(n^3)$ to $\Theta(n^{\log_2(7)})$, which is approximately $\Theta(n^{2.81})$. This improvement in time complexity makes it an efficient algorithm for large matrix multiplications.

4 Derivation of $T(n) = O(n^2)$

We will derive that $T(n) = O(n^2)$ for the given expression:

$$T(n) = c \cdot n^2 + c\left(\frac{3}{4}n\right)^2 + c\left(\frac{3}{4^2}n\right)^2 + c\left(\frac{3}{4^3}n\right)^2 + \ldots + c\left(\frac{3}{4^{(\log_4 n - 1)}}n\right)^2 + \Theta(n^{\log_4 3})$$

First, rewrite the expression:

$$T(n) = c \cdot n^2 + c \cdot \left(\frac{3^2}{4^2}n^2\right) + c \cdot \left(\frac{3^3}{4^3}n^2\right) + \ldots + c \cdot \left(\frac{3^{(\log_4 n - 1)}}{4^{(\log_4 n - 1)}}n^2\right) + \Theta(n^{\log_4 3})$$

Factor out constants:

$$T(n) = c \cdot n^2 \sum_{k=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^k + \Theta(n^{\log_4 3})$$

The summation is a geometric series:

$$\sum_{k=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^k = \frac{1 - \left(\frac{3}{4}\right)^{\log_4 n}}{1 - \frac{3}{4}}$$

Simplify:

$$\sum_{k=0}^{\log_4 n - 1} \left(\frac{3}{4}\right)^k = 4(1 - n^{\log_4 3 - 1})$$

Substitute back into $T(n)$:

$$T(n) = 4c \cdot n^2 - 4c \cdot n^{\log_4 3} + \Theta(n^{\log_4 3})$$

Since $n^{\log_4 3}$ is the dominant term, $T(n) = O(n^{\log_4 3})$.
Case 1: $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$
In this case, we have $f(n) = O(n^{\log_b a - \varepsilon})$, which means $f(n)$ is polynomially smaller than $n^{\log_b a}$.
According to the Master Theorem, we are in **Case 1**:

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

Now, we'll compare $f(n)$ with $n^{\log_b a}$:

$$f(n) = O(n^{\log_b a - \varepsilon}) \quad \text{and} \quad n^{\log_b a} = n^{\log_a a} = n$$

Since $f(n)$ is polynomially smaller than $n^{\log_b a}$, we are in **Case 1**.
Case: $f(n) = O(n^{\log_a b + \epsilon})$
In this case, we have $f(n) = O(n^{\log_a b + \epsilon})$ for some $\epsilon > 0$.
According to the Master Theorem, we are in **Case 3** (a variation of Case 3):

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

Now, we'll compare $f(n)$ with $n^{\log_a b + \epsilon}$:

$$f(n) = O(n^{\log_a b + \epsilon}) \quad \text{and} \quad n^{\log_a b + \epsilon} = n^{\log_a b} \cdot n^{\epsilon}$$

Since $f(n)$ is bounded above by $n^{\log_a b + \epsilon}$, and $\epsilon > 0$, we can conclude that $f(n)$ dominates, and thus:

$$T(n) = O(f(n))$$

So, in this case, $T(n) = O(f(n))$.

Theorem 1 *Let $T(n)$ be defined as:*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & \text{if } n > 1 \end{cases}$$

where $a = b^i$. Then, for $a > 1$ and $b > 1$, $T(n)$ has the following asymptotic behavior:

1. *If* $f(n) = \Theta(1)$ *when* $n = 1$, *then* $T(n) = \Theta(1)$ *for all* n.
2. *If* $f(n) = \Theta(n^{\log_b a})$, *then* $T(n) = \Theta(n^{\log_b a})$.
3. *If* $f(n) = \Theta(n^{\log_b a} \log^k n)$ *for some* $k > 0$, *then* $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
4. *If* $f(n) = \Theta(n^c)$ *for some* $c > \log_b a$, *then* $T(n) = \Theta(f(n))$.

Algorithm 1 Heap Sort

```
1: procedure HeapSort(A)
2:     n ← length of A
3:     for i ← ⌊n/2⌋ down to 1 do
4:         Heapify(A, n, i)
5:     end for
6:     for i ← n down to 2 do
7:         Swap A[1] and A[i]
8:         Heapify(A, i − 1, 1)
9:     end for
10: end procedure
11: procedure Heapify(A, n, i)
12:     largest ← i
13:     left ← 2 · i
14:     right ← 2 · i + 1
15:     if left ≤ n and A[left] > A[largest] then
16:         largest ← left
17:     end if
18:     if right ≤ n and A[right] > A[largest] then
19:         largest ← right
20:     end if
21:     if largest ≠ i then
22:         Swap A[i] and A[largest]
23:         Heapify(A, n, largest)
24:     end if
25: end procedure
```