Analysis of Algorithms I: Strassen's Algorithm and the Master Theorem

Xi Chen

Columbia University

BinarySearch: Find a number in a sorted sequence

BinarySearch (A, x), where $A = (a_1, a_2, \ldots, a_n)$ is nondecreasing

1. If $n = 1$ then output 1 if $a_1 = x$; and output nil otherwise
2. Compare x with $a_{n/2}$
3. Case $x = a_{n/2}$: output $n/2$
4. Case $x > a_{n/2}$: output BinarySearch $((a_{n/2+1}, \ldots, a_n), x)$
5. Case $x < a_{n/2}$: output BinarySearch $((a_1, \ldots, a_{n/2-1}), x)$

The running time $T(n)$ of BinarySearch is characterized by:

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(n/2) + \Theta(1) \quad \text{for } n \geq 2 \end{aligned}$$

A little sloppy here: Should be $T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$. But this will not affect the order of $T(n)$ (as we saw from the last class), and will be further justified by the Master theorem later.

We now use the substitution method (Section 4.3 of the textbook) to solve the recurrence. First, spell out the constants:

$$\begin{aligned} T(1) &= c_1 \\ T(n) &= T(n/2) + c_2 \quad \text{for } n \geq 2 \end{aligned}$$

Then make a good guess: Here we show that for some positive constants a and b to be specified later,

$$T(n) \leq a \lg n + b \tag{1}$$

for all n being powers of 2. The proof uses induction.

Proof.

1. **Basis**: We know $T(1) = c_1$. On the other hand, when $n = 1$, $a \lg n + b = b$. So if we set b to be any positive constant $\geq c_1$ (e.g., set $b = c_1$), (1) holds for $n = 1$.
2. **Induction Step**: Assume (1) holds for $2^0, 2^1, \ldots, 2^{k-1}$, for some $k \geq 1$. We show (1) also holds for $n = 2^k$. To this end,

$$T(n) = T(2^{k-1}) + c_2 \leq a(k-1) + b + c_2 = ak + b + (c_2 - a)$$

   As a result, $T(n) \leq ak + b$ if we set a to be any positive constant $\leq c_2$ (e.g., set $a = c_2$).
3. By setting $a = c_2$ and $b = c_1$, we conclude from induction that $T(n) \leq a \lg n + b$ for all $n = 1, 2, 4, \ldots$ $\square$

As a result, we have $T(n) \leq a \lg n + b = O(\lg n)$. One weakness of the substitution method is that it is important to make a good guess. For example, if we guess that $T(n) \leq an$ for some positive constant a, then the whole proof would still go through for some appropriate a (because this claim IS CORRECT), even though the bound $O(n)$ is very loose indeed. So always try to apply the Master theorem first. Use the substitution method only when the Master theorem does not apply.

Powering a number: Given a and n, compute a^n.

Power (a, n)

1. If $n = 1$, output a
2. If n is even, $b =$ Power $(a, n/2)$ and output b^2
3. If n is odd, $b =$ Power $(a, (n-1)/2)$ and output $a \cdot b^2$

The running time $T(n)$ is described by the same recurrence:

$$T(1) = \Theta(1)$$
$$T(n) = T(n/2) + \Theta(1) \quad \text{for } n \geq 2$$

So we conclude that $T(n) = O(\lg n)$, while the brute force algorithm takes $(n-1)$ multiplications.

Matrix multiplication: Given two $n \times n$ matrices $\mathbf{A} = (a_{i,j})$ and $\mathbf{B} = (b_{i,j})$, $1 \leq i, j \leq n$, compute $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, where $\mathbf{C} = (c_{i,j})$ and

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}, \qquad \text{for all } i, j : 1 \leq i, j \leq n$$

To compute each $c_{i,j}$ using the equation above, it takes n multiplications and $(n-1)$ additions. So the running time is

$$n^2 \cdot \Theta(n) = \Theta(n^3)$$

Can we use Divide-and-Conquer to speed up?

Denote **A** and **B** by

$$\mathbf{A} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad \text{and} \quad \mathbf{B} = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

where all $A_{i,j}$ and $B_{i,j}$ are $n/2 \times n/2$ matrices. If we denote

$$\mathbf{A} \cdot \mathbf{B} = \mathbf{C} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

where all $C_{i,j}$ are $n/2 \times n/2$ matrices, then we have

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$
$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$
$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

This suggests the following Divide-and-Conquer algorithm:

MM$(\mathbf{A}, \mathbf{B})$, where $\mathbf{A}$ and $\mathbf{B}$ are both $n \times n$ matrices

1. If $n = 1$, output $a_{1,1} \cdot b_{1,1}$
2. Compute MM$(A_{1,1}, B_{1,1})$ + MM$(A_{1,2}, B_{2,1})$
3. Compute MM$(A_{1,1}, B_{1,2})$ + MM$(A_{1,2}, B_{2,2})$
4. Compute MM$(A_{2,1}, B_{1,1})$ + MM$(A_{2,2}, B_{2,1})$
5. Compute MM$(A_{2,1}, B_{1,2})$ + MM$(A_{2,2}, B_{2,2})$

The running time $T(n)$ of MM (for multiplying two $n \times n$ matrices) is then described by the following recurrence:

$$
\begin{aligned}
T(1) &= \Theta(1) \\
T(n) &= 8 \cdot T(n/2) + \Theta(n^2) \qquad \text{for } n \geq 2
\end{aligned}
$$

because we make 8 recursive calls (for multiplying $n/2 \times n/2$ matrices), and a constant many (4 indeed) matrix additions when combining the solutions. Unfortunately, solving the recurrence using the Master theorem gives us $T(n) = \Theta(n^3)$, where 3 comes from $\log_2 8$. Can we use less multiplications and do better than n^3?

In Strassen's algorithm, the following 7 $n/2 \times n/2$ matrices $P_1, \ldots, P_7$ are computed first using 7 recursive calls:

$$P_1 = A_{1,1} \cdot (B_{1,2} - B_{2,2})$$
$$P_2 = (A_{1,1} + A_{1,2}) \cdot B_{2,2}$$
$$P_3 = (A_{2,1} + A_{2,2}) \cdot B_{1,1}$$
$$P_4 = A_{2,2} \cdot (B_{2,1} - B_{1,1})$$
$$P_5 = (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2})$$
$$P_6 = (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2})$$
$$P_7 = (A_{1,1} - A_{2,1}) \cdot (B_{1,1} + B_{1,2})$$

Then it uses additions and subtractions to get $C_{i,j}$:

$$\begin{aligned}
C_{1,1} &= P_5 + P_4 - P_2 + P_6 \\
C_{1,2} &= P_1 + P_2 \\
C_{2,1} &= P_3 + P_4 \\
C_{2,2} &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

It can be verified that the magic cancelations result in exactly the same $C_{i,j}$'s in $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$.

The running time $T(n)$ is now described by

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= 7 \cdot T(n/2) + \Theta(n^2) \quad \text{for } n \geq 2 \end{aligned}$$

because we only make 7 recursive calls instead of 8, and use 18 (still a constant though) matrix additions, 10 before the recursive calls and 8 after. Solving this recurrence using the Master theorem, we get $T(n) = \Theta(n^{\lg 7}) = \Theta(n^{2.81\ldots})$.

Finally, we describe the Master theorem. Let $a \geq 1$ and $b > 1$ be constants. We are interested in $T(n)$ described by:

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= a \cdot T(n/b) + f(n) \quad \text{for } n \geq 2 \end{aligned}$$

A little sloppy here: n/b should be interpreted as either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$, though this will not change the conclusions of the three cases to be discussed. (For the proof dealing with floors and ceilings, check Section 4.6.2 of the textbook.) In what follows, we consider $T(\cdot)$ over powers of b: $n = 1, b, b^2, \ldots$

A key constant in solving the recurrence is $t = \log_b a$ with

$$b^t = a$$

Let $n = b^k$, where $k = \log_b n$. First, from the recursion tree generated using $T(n) = a \cdot T(n/b) + f(n)$ in Fig 4.7 (Page 99) of the textbook, we have

$$T(n) = a^k \cdot T(1) + \sum_{i=0}^{k-1} a^i \cdot f(n/b^i)$$

where $a^k \cdot T(1)$ is the contribution from the leaves, and $a^i \cdot f(n/b^i)$ is the contribution from nodes on level i, $i = 0, 1, \ldots, k-1$. Since $T(1) = \Theta(1)$, the contribution of the leaves is $\Theta(a^k) = \Theta((b^t)^{\log_b n}) = \Theta(n^t)$.

Theorem

Case 1 of the Master theorem: If $f(n) = O(n^{t-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^t)$.

This case basically says that if $f(n)$ is smaller than n^t, then $T(n) = \Theta(n^t)$. In the next slide, we show that in this case, the total contribution from levels $0, 1, \ldots, k-1$ is no more than the contribution from the leaves. As a result, $T(n) = \Theta(n^t)$, where the contribution from the leaves is the dominating term.

To see this, we plug in $f(n) = O(n^{t-\epsilon})$ and the total contribution from levels $0, 1, \ldots, k-1$ is the following sum

$$\sum_{i=0}^{k-1} a^i \cdot f(n/b^i) = \Theta\left(\sum_{i=0}^{k-1} a^i \cdot (n/b^i)^{t-\epsilon}\right)$$

Focusing on the sum inside the Θ, it becomes

$$n^{t-\epsilon}\sum_{i=0}^{k-1}(ab^{\epsilon}/b^t)^i = n^{t-\epsilon}\sum_{i=0}^{k-1}(b^{\epsilon})^i = n^{t-\epsilon}\frac{b^{\epsilon k}-1}{b^{\epsilon}-1} = n^{t-\epsilon}\frac{n^{\epsilon}-1}{b^{\epsilon}-1} = O(n^t)$$

where the first equation uses $a = b^t$ and the second uses the geometric series. As a result, $T(n) = \Theta(n^t) + O(n^t) = \Theta(n^t)$.

Example of Case 1: In the recurrence of Strassen's algorithm:

$$a = 7 \quad b = 2 \quad \text{and} \quad f(n) = \Theta(n^2)$$

Therefore, $t = \log_2 7 = 2.81$ and it is clear that $f(n) = O(n^{t-\epsilon})$ if we set ϵ to be 0.1. As a result, Case 1 of the Master theorem applies, and we conclude that $T(n) = \Theta(n^t) = \Theta(n^{\lg 7})$.

Theorem

Case 2 of the Master theorem: If $f(n) = \Theta(n^t)$, $T(n) = \Theta(n^t \lg n)$.

This case basically says that if $f(n)$ is of the same order as n^t, then $T(n) = \Theta(n^t \lg n)$. In the next slide, we show that in this case, the contribution from each level i, $i = 0, 1, \ldots, k-1$ is $\Theta(n^t)$. As a result, we have

$$T(n) = (k+1) \cdot \Theta(n^t) = \Theta(n^t \log_b n) = \Theta(n^t \lg n)$$

where the last equation follows from $\lg n = \Theta(\log_b n)$.

To see this, the contribution from level i is

$$a^i \cdot f(n/b^i) = a^i \cdot \Theta((n/b^i)^t) = \Theta(a^i \cdot n^t/b^{ti}) = \Theta(n^t)$$

because $b^t = a$. Case 2 then follows.

Example of Case 2: In the recurrence of Merge Sort:

$$a = b = 2 \quad \text{and} \quad f(n) = \Theta(n)$$

Thus, $t = \log_b a = 1$ and $f(n) = \Theta(n^t)$. So Case 2 applies and we conclude that $T(n) = \Theta(n^t \lg n) = \Theta(n \lg n)$.

Also in the recurrence of Binary Search:

$$a = 1 \quad b = 2 \quad \text{and} \quad f(n) = \Theta(1)$$

Thus, $t = \log_b a = 0$ and $f(n) = \Theta(n^t) = \Theta(1)$. So Case 2 applies and we conclude that $T(n) = \Theta(n^t \lg n) = \Theta(\lg n)$.

Theorem

Case 3 of the Master theorem: If $f(n) = \Omega(n^{t+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

This case basically says that if $f(n)$ is larger than n^t and satisfies a regularity condition, then $T(n) = \Theta(f(n))$. The proof can be found in the textbook. In this case, the contribution from level 0, $f(n)$, dominates the total contribution from levels $1, 2, \ldots, k-1$ as well as the leaves.

Example of Case 3: $T(n) = 3 \cdot T(n/2) + n^2$, where

$$a = 3 \quad b = 2 \quad \text{and} \quad f(n) = \Theta(n^2)$$

Therefore, $t = \log_2 3 = 1.58\ldots$ and $f(n) = \Omega(n^t + \epsilon)$ if we set $\epsilon = 0.1$. Also $f(n)$ satisfies the regularity condition:

$$af(n/b) = 3(n/2)^2 = (3/4)n^2 = (3/4)f(n)$$

So Case 3 applies and $T(n) = \Theta(f(n)) = \Theta(n^2)$.

To conclude, the Master theorem compares the order of $f(n)$ with n^t where $t = \log_b a$, and solves the recurrence depending on which one is larger.