CS 577 Introduction to Algorithms: Strassen's Algorithm and the Master Theorem

Jin-Yi Cai

University of Wisconsin–Madison

In the last class, we described InsertionSort and showed that its worst-case running time is $\Theta(n^2)$.
Check Figure 2.2 for the intuition why it is correct. To give a formal proof, we use (mathematical) induction.

For InsertionSort, we prove the following theorem:

Theorem

Let $n \geq 1$ be a positive integer. If $A = (a_1, \ldots, a_n)$ is the input sequence of InsertionSort, then after the jth loop, where $j = 1, 2, \ldots, n$, the sublist $A[1 : j]$ of length j and is a nondecreasing permutation of the first j integers of the original A.

The intuition of this statement comes from the example of Figure 2.2 in the textbook. The correctness of InsertionSort clearly follows from this theorem.

Proof.

1. **Basis**: The statement holds for $A[1:1]$ obviously.
2. **Induction Step**: Assume the statement holds for some $k : 1 \leq k < n$. So after k steps, $A[1:k]$ is a permutation of $a_1, \ldots, a_k$ in nondecreasing order. Now in the $(k+1)$th iteration of the loop, InsertionSort finds the first element in $A[1:k]$ smaller than a_{k+1} and inserts a_{k+1} after that integer. (If no such element exists, i.e., a_{k+1} is the minimum value among $A[1:k+1]$, then it is inserted at the beginning.) As a result, after $k+1$ steps, $A[1:k+1]$ is a permutation of $a_1, \ldots, a_{k+1}$ in nondecreasing order.
3. Conclude that the statement holds for $1, 2, \ldots, n$.

Next we describe MergeSort, a sorting algorithm that is asymptotically faster than InsertionSort. It is an example of the Divide-and-Conquer technique:

1. Divide the problem into smaller subproblems
2. Conquer (or solve) each of the subproblems recursively
3. Combine the solutions to the subproblems to get a solution to the original problem

MergeSort (A), where $A = (a_1, \ldots, a_n)$ is a sequence of n integers

1. If $n = 1$, return
2. MergeSort $(a_1, a_2, \ldots, a_{\lceil n/2 \rceil})$
3. MergeSort $(a_{\lceil n/2 \rceil + 1}, \ldots, a_{n-1}, a_n)$
4. Merge the two sequences obtained to produce a sorted permutation of the n integers in A

An implementation of line 4 can be found on page 31 of the textbook. It takes time $\Theta(n)$ (or in other words, cn steps for some positive constant c). And one can use induction (page 32 and 33) to show that, if the two sequences we obtain from the two recursive calls are sorted sequences of $a_1, \ldots, a_{\lceil n/2 \rceil}$ and $a_{\lceil n/2 \rceil + 1}, \ldots, a_n$, respectively, then the Merge subroutine outputs a sorted permutation of $a_1, \ldots, a_n$.

Practice the use of induction to show that

Theorem

MergeSort outputs correctly for sequences of length $n = 1, 2, \ldots$

To understand the performance of MergeSort, we use $T(n)$ to denote the number of steps it takes over sequences of length n. We get the following recurrence:

$$T(1) = \Theta(1) \qquad \text{usually let } \Theta(1) \text{ denote a positive constant}$$

$$T(n) = \Theta(1) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \qquad \text{for all } n \geq 2$$

For now, we focus the analysis on powers of 2: $n = 2^k$ for some integer $k \geq 0$. We will see later that this suffices to understand the asymptotic complexity of $T(n)$. For powers of 2, we have

$$T(1) = \Theta(1)$$

$$T(n) = 2T(n/2) + \Theta(n)$$

Putting the constants back gives us

$$T(1) = c_1 \tag{1}$$

$$T(n) = 2T(n/2) + c_2 n \tag{2}$$

for some positive constants c_1 and c_2.

Let $n = 2^k$. Then start with $T(n)$ and substitute it using (2):

$$T(n) = 2T(n/2) + c_2 n$$

Next, expand it further by substituting $T(n/2)$ using (2): $T(n/2) = 2T(n/4) + c_2 n/2$. We get

$$T(n) = 2(2T(n/4) + c_2 n/2) + c_2 n = 4T(n/4) + c_2 n + c_2 n$$

Repeat it for k rounds, and we get the so-called recursion tree in Fig 2.5 on page 38. $T(n)$ is the sum of all numbers on the nodes:

$$T(n) = nT(1) + c_2 nk = c_2 n \lg n + c_1 n$$

Intuitively you can visualize, when after $k = \log_2 n$ steps, the quantity in front of $T(1)$ has grown to be $2^k = n$.

But the dominating terms is $n \lg n$.

To see why this gives us $T(n) = \Theta(n \lg n)$, we use the fact that $T(n)$ is monotonically nondecreasing over $n = 1, 2, \ldots$ (Try to prove it using induction). Given any integer $n \geq 1$, we let n' denote the largest power of 2 that is smaller than n. This implies that $n' < n \leq 2n'$ and by the monotonicity of T, we have

$$T(n') \leq T(n) \leq T(2n')$$

Because both n' and $2n'$ are powers of 2, from the last slide,

$$T(n) \leq T(2n') = 2c_2 n' \lg(2n') + 2c_1 n' < 2c_2 n \lg(2n) + 2c_1 n$$

$$T(n) \geq T(n') = c_2 n' \lg n' + c_1 n' \geq c_2 (n/2) \lg(n/2) + c_1 (n/2)$$

From these two inequalities, it is easy to show $T(n) = \Theta(n \lg n)$.

From the analysis, we see that c_1 and c_2 do not affect the asymptotic complexity of $T(n)$. This is why we can suppress them and denote the running time of line 1 and 4 by $\Theta(1)$ and $\Theta(n)$, respectively, and we do not care what they are exactly in the analysis.

However, the coefficient 2 of $T(n/2)$ cannot be suppressed. Changing it would change the order of $T(n)$. In particular, if we change it to 3, then the recursion tree would look different: every internal node would have 3 children instead of 2. This would change the order of $T(n)$ significantly. We will use an example: Strassen's algorithm for matrix multiplication, to demonstrate the importance of this constant in the next class.

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 $((a_1, \ldots, a_{n/2-1}), x)$

Technically we must use floor ... or ceiling ...

Compare x with $a_{\lfloor n/2 \rfloor}$, etc.

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 $((a_1, \ldots, a_{n/2-1}), x)$

Technically we must use floor ... or ceiling ...

Compare x with $a_{\lfloor n/2 \rfloor}$, etc.

Note that, after line 1, we have $n \geq 2$. So we would not have underflow $\lfloor n/2 \rfloor = 0$.

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

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

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 earlier), 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:

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

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

$$T(n) \leq a \lg n + b \tag{3}$$

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 $b = c_1$), (3) holds for $n = 1$.
2. **Induction Step**: Assume (3) holds for 2^{k-1}, for some $k \geq 1$. We show (3) 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 $\geq c_2$ (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 $n = 1, 2, 4, \ldots$

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 far from being tight.

The Master Theorem, to be described, gives a mechanical method to get a good bound, without the need to guess the right answer.

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 $a \cdot b^2$

One can consider the arithmetic operations to be modulo some integer m.

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.

It is worthwhile to think about where did the saving come from.
Consider a^{16}.
Consider $a^{23} = 2^{16+0\cdot 8+4+2+1}$.

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 it up?

Denote $\mathbf{A}$ and $\mathbf{B}$ 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:

$\mathrm{MM}(\mathbf{A}, \mathbf{B})$, 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 $\mathrm{MM}(A_{1,1}, B_{1,1}) + \mathrm{MM}(A_{1,2}, B_{2,1})$
3. Compute $\mathrm{MM}(A_{1,1}, B_{1,2}) + \mathrm{MM}(A_{1,2}, B_{2,2})$
4. Compute $\mathrm{MM}(A_{2,1}, B_{1,1}) + \mathrm{MM}(A_{2,2}, B_{2,1})$
5. Compute $\mathrm{MM}(A_{2,1}, B_{1,2}) + \mathrm{MM}(A_{2,2}, B_{2,2})$

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

$$T(1) = \Theta(1)$$
$$T(n) = 8 \cdot T(n/2) + \Theta(n^2) \quad \text{for } n \geq 2$$

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 seven(!) $n/2 \times n/2$ matrices $P_1, \ldots, P_7$ are computed first using 7 recursive calls:

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

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

$$C_{1,1} = P_5 + P_4 - P_2 + P_6$$
$$C_{1,2} = P_1 + P_2$$
$$C_{2,1} = P_3 + P_4$$
$$C_{2,2} = P_5 + P_1 - P_3 - P_7$$

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

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

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:

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

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: $n = 1, b, b^2, \ldots$

Look at this formula:

$$a^{\log_b n} = n^{\log_b a}$$

To see this, just take $\log_b$ on both sides.

Call this quantity on the exponent of n:

$$t = \log_b a$$

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 $\Theta(a^k) = \Theta((b^t)^{\log_b n}) = \Theta(n^t)$.

$$T(n) = a \cdot T(n/b) + f(n) \quad \text{and} \quad t = \log_b a$$

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 the total cost is dominated by the leaves: $T(n) = \Theta(n^t)$. 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 $O(n^t)$:

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

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 each level of the recursion tree contributes roguhly the same amount, with a total $T(n) = \Theta(n^t \lg n)$. 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, the top level with value $f(n)$, 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.