A Lecture on Divide-and-Conquer Algorithms and the Master Theorem

Adam Wertheimer, Jason Pizzuco. McGill University.

February 1, 2018

This is the augmented transcript of a lecture given by Luc Devroye on the 16th of January 2018 for a Data Structures and Algorithms class (COMP-252, McGill University). The subject was various divide-and-conquer algorithms and their recurrence relations.

Recurrences

Definition 1. Given a problem with input size n and an algorithm that solves this problem in time T_n (where T_n is a function of n), a **recurrence** or a **recurrence relation** is a formula for T_n in terms of any of the previous running times $\{T_{n-1}, T_{n-2}, \ldots, T_1, T_0\}$ and n.

Example 2. Consider the following formula for T_n:

$$\begin{cases} T_n = 2T_{n-1} + T_{n-2} + n^2 \;\; \forall n \geq 2, \\ T_1 = 1, \; T_0 = 0. \end{cases} \tag{1}$$

The first line of equation (1) represents a recurrence for T_n that depends on the previous two running times T_{n-1} and T_{n-2} and the input size n, as long as n is greater or equal to 2. The second line represents the base case running times when $n = 1$ or $n = 0$.

Recall the following recurrences below:

Chip Testing (n is the number of chips) :

$$\begin{cases} T_n \leq 3n/2 + T_{\lfloor n/2 \rfloor}, \\ T_0 = T_1 = 0. \end{cases}$$

Fast Exponentiation (n is the exponent) :

$$\begin{cases} T_n \leq 1 + T_{\lfloor n/2 \rfloor}, & \text{(RAM Model)} \\ T_n \leq n^2 + T_{\lfloor n/2 \rfloor}, & \text{(Bit Model)} \\ T_0 = T_1 = 0. \end{cases}$$

Merge Sort (n is the list size) :

$$\begin{cases} T_n \leq T_{\lfloor n/2 \rfloor} + T_{\lceil n/2 \rceil} + n - 1, \\ T_0 = T_1 = 0. \end{cases}$$

Binary Search (n is the sorted list size) :

$$\begin{cases} T_n \leq 1 + T_{\lfloor n/2 \rfloor}, \\ T_0 = 0, \; T_1 = 1. \end{cases}$$

Karatsuba Multiplication (n-bit numbers) :

$$\begin{cases} T_n \leq 3T_{n/2} + n, \\ T_0 = 0, \; T_1 = 1. \end{cases}$$

Toom-3 Multiplication (n-bit numbers) :

$$\begin{cases} T_n \leq 5T_{n/3} + n, \\ T_0 = 0, \; T_1 = 1. \end{cases}$$

Mathematical Induction

Definition 3. Given a collection of statements $\{S_n\}_{n \geq n_0}$ indexed by the set of integers greater or equal to some $n_0 \in \mathbb{N}$, we can prove S_n for all $n \geq n_0$ using the three steps of **mathematical induction**:

1. We show that the **base case** statement S_{n_0} holds,
2. We assume the **inductive hypothesis**, which is that the statement S_k holds for all k such that $n_0 \leq k < n$,
3. We prove the **inductive step**, which is that the statement S_n holds given the assumption of the inductive hypothesis.

Mathematical induction is analogous to the act of climbing a ladder. To climb a ladder, one starts by climbing the first rung (base case). Next, one must reach the first $n-1$ rungs (inductive hypothesis) and then the n^{th} rung (inductive step). Since this holds for all $n \in \mathbb{N}$, all of the ladder's rungs can be reached (end of proof).

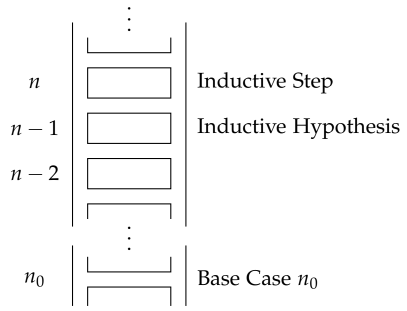


Figure 1: The "climbing a ladder" analogy for mathematical induction.

Chip-Testing Algorithm

Example 4. Show that in the chip testing algorithm, we have $T_n \leq C \cdot n$ for all n and some $C > 0$.[1]

[1] Recall that an algorithm has worst-case running time $T_n = O(n)$ if

$$\exists C \in \mathbb{R} \text{ such that } T_n \leq C \cdot n \;\; \forall n \geq n_0,$$

where T_{n_0} is some base case run time and $n_0 > 0$.

The **base case** $n = 0$ is trivial, since $0 = T_0 \leq C \cdot n$ for *any* $C \in \mathbb{R}$. Now, assume the **inductive hypothesis**, which states that

$$T_k \leq C \cdot k, \text{ for all } k \in \{0, 1, 2, \ldots, n-1\}.$$

By hypothesis and definition of T_n, we have the following:

$$T_n \leq \frac{3n}{2} + T_{\lfloor n/2 \rfloor} \leq \frac{3n}{2} + C \cdot \left\lfloor \frac{n}{2} \right\rfloor \leq \frac{3n}{2} + C \cdot \frac{n}{2},$$

but we want to prove the **inductive step** or $T_n \leq C \cdot n$, so we will need

$$\frac{3n}{2} + C \cdot \frac{n}{2} \overset{?}{\leq} C \cdot n,$$

which holds if $C \geq 3$. Therefore, $T_n \leq 3 \cdot n$ for all n. □

Note that whenever we place a question mark on top of a relational operator such as $\leq$ or $=$, the relation is simply one that we *hope* holds.

Binary Search

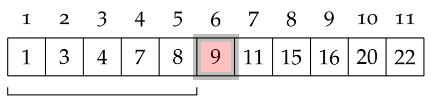
Definition 5. Given a sorted list $[x_1, x_2, \ldots, x_n]$ of size n and an object x', the divide-and-conquer **Binary Search** algorithm returns the list position of the object in question (if it exists in the list).

The algorithm uses a **ternary oracle**, which takes two objects x_a, x_b as input and returns one of $\{(x_a < x_b), (x_a = x_b), (x_a > x_b)\}$.

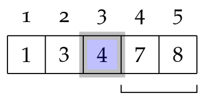
- If the list $[\;]$ is of size 0, then it does not contain the desired object.
- If the list $[x]$ is of size 1 and the oracle returns $x' = x$, then the lone element in the list is our desired object. If not, then the list does not contain the desired object.

The algorithm uses the fact these lists are sorted to its advantage. It repeatedly compares the desired object to the middle element of the list and calls itself on the corresponding half of the list until the object is found or a base case is reached.

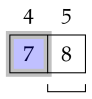
- If n is even, we use $x_{n/2}$ as the middle object. If the oracle returns $x' = x_{n/2}$ then we have found the object. If not, the list is split into $[x_1, x_2, \ldots, x_{(n/2)-1}]$ and $[x_{(n/2)+1}, x_{(n/2)+2}, \ldots, x_n]$, and we call Binary Search on the sub-list that may contain x'.
- If n is odd, we use $x_{(n+1)/2}$ as the middle object. If the oracle returns $x' = x_{(n+1)/2}$ then we have found the object. If not, the list is split into $[x_1, x_2, \ldots, x_{((n-1)/2)-1}]$ and $[x_{((n+1)/2)+1}, x_{((n+1)/2)+2}, \ldots, x_n]$, and we call Binary Search on the sub-list that may contain x'.



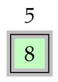
Here, the middle element (9) lies in the 6th slot. The oracle returns $8 < 9$, so we call Binary Search on the left sub-list.



Here, the middle element (4) lies in the 3rd slot. The oracle returns $8 > 4$, so we call Binary Search on the right sub-list.



Here, the middle element (7) lies in the 1st slot. The oracle returns $8 > 7$, so we call Binary Search on the right sub-list.



We have arrived at the base case $n = 1$. The oracle returns $8 = 8$ so the algorithm returns the index 5.

Figure 2: Finding the number 8 in a sorted list using Binary Search

Example 6. Show that for Binary Search, the worst-case time T_n satisfies $T_n = O(\log_2(n))$.

The case $n = 0$ of Binary Search requires no oracle calls, so $T_0 = 0$. When $n = 1$, a single oracle call is required, so $T_1 = 1$. Calling Binary Search on a list of size $n \geq 2$ requires a single oracle call and perhaps a Binary Search call on a list of size $\lfloor n/2 \rfloor$, so:

$$\begin{cases} T_n \leq 1 + T_{\lfloor n/2 \rfloor} \quad \forall n \geq 2, \\ T_0 = 0,\ T_1 = 1. \end{cases}$$

We will show that $T_n \leq 1 + C \cdot \log_2(n)$ for some $C > 0$.

The **base case** $n = 1$ is trivial, since $T_1 = C \cdot \log_2(1)$ for *any* $C \in \mathbb{R}$. Now, assume the **inductive hypothesis**, which states that:

$$T_k \leq C \cdot \log_2(k) + 1, \text{ for all } k \in \{1, 2, \ldots, n-1\}.$$

By hypothesis and definition of T_n, we have the following:

$$\begin{aligned} T_n \leq 1 + T_{\lfloor n/2 \rfloor} &\leq 1 + C \cdot \log_2 \lfloor n/2 \rfloor + 1 \\ &\leq 1 + C \cdot \log_2 (n/2) + 1 \\ &= 1 + C \cdot \log_2(n) - C \cdot \log_2(2) + 1 \\ &= 1 + C \cdot \log_2(n) + 1 - C \\ &\leq 1 + C \cdot \log_2(n) \text{ whenever } C \geq 1. \end{aligned}$$

So we have shown that $T_n \leq 1 + \log_2(n)$ holds for all n. □

Exercise 7. Show by induction that if Binary Search runs with a binary oracle that returns one of $\{(x_a \leq x_b), (x_a \not\leq x_b)\}$ then:

Hint: The algorithm remains the same, but two oracle calls are necessary to show that two objects are equal.

$$\begin{cases} T_n \leq 2 + \lceil \log_2(n) \rceil, \\ T_0 = 0,\ T_1 = 2. \end{cases}$$

Master Theorem

We have seen many algorithms that have recurrences of the form

$$T_n = a \cdot T_{n/b} + f(n),$$

for some integers a and b such that $a \geq 1$ and $b > 1$ along with a positive function $f(n)$ of n. We can prove that $T_n = \mathcal{O}(g(n))$ inductively, but we need to conjecture a guess for $g(n)$ ahead of time. The Master Theorem gives us a way of hazarding a guess for $g(n)$.

Theorem 8 (Master Theorem). *Given a recurrence of the form*

$$T_n = a \cdot T_{\lfloor n/b \rfloor} + f(n), \tag{2}$$

(a) If there exists some $\epsilon > 0$ and some $n_0 \in \mathbb{N}$ such that [2]

$$n^{\log_b(a)}/f(n) > n^{\epsilon} \quad \forall n \geq n_0,$$

then $T_n = O(n^{\log_b(a)})$.

(b) If there exists some $\epsilon > 0$ and some $n_0 \in \mathbb{N}$ such that [3]

$$f(n)/n^{\log_b(a)} > n^{\epsilon} \quad \forall n \geq n_0,$$

then $T_n = O(f(n))$.

(c) If $f(n) = \Theta(n^{\log_b(a)})$ then $T_n = \Theta(n^{\log_b(a)} \cdot \log_b(n))$.

[2] If we have equality, meaning that

$$n^{\log_b(a)}/f(n) = \Theta(n^c) \;\; \forall n \geq n_0,$$

then $T_n = \Theta(n^{\log_b(a)})$ instead.

[3] This case holds under the technical assumption

$$\liminf_{n\to\infty} \left(\frac{f(n)}{a \cdot f(n/b)} \right) > 1.$$

Example 9. Approximating the time complexity for Merge Sort yields

$$T_n \leq T_{\lfloor n/2 \rfloor} + T_{\lceil n/2 \rceil} + n - 1 \approx 2 \cdot T_{n/2} + n - 1, \tag{3}$$

which is equation (2) with $a = b = 2$ and $f(n) = n - 1$. We have

$$f(n) = n - 1 = \Theta(n) = \Theta(n^{\log_2(2)}) = \Theta(n^{\log_b(a)}),$$

so the third case applies here. Therefore:

$$T_n = \Theta(n^{\log_2(2)} \cdot \log_2(n)) = \Theta(n \cdot \log_2(n)).$$

One can show formally by induction that $T_n \leq n \cdot \log_2(n)$ for all n.

The recursive relation given by equation (3) holds for $n \geq 2$, since the base cases $T_0 = T_1 = 0$ are handled separately. This implies that $f(n) = n - 1 \geq 1 > 0$ is a positive function of n and so we can apply the Master Theorem.

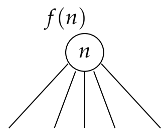
Exercise 10. Solve the recurrences below using the Master Theorem:

Chip Testing	$O(n)$
Fast Exponentiation (RAM Model)	$O(\log(n))$
Fast Exponentiation (Bit Model)	$O(n^2)$
Merge Sort	$\Theta(n \log(n))$
Karatsuba Multiplication	$O(n^{\log_2(3)}) \approx O(n^{1.585})$
Toom-3 Multiplication	$O(n^{\log_3(5)}) \approx O(n^{1.465})$

Recursion Trees

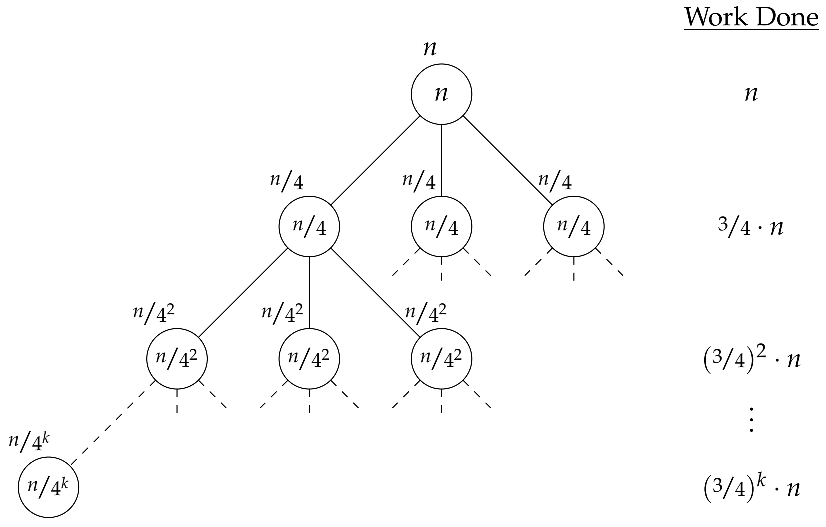
Definition 11. Given a recurrence of the form $T_n = a \cdot T_{n/b} + f(n)$, calling T_n costs $f(n)$ units of "work" and results in a new problem with worst-case time $T_{n/b}$ being called "a" times. To visualize this scenario, we define an a-ary tree aptly named the algorithm's **recursion tree**. Each node represents an algorithm call, with the root node referring to the first call on a problem of size n. Given the recursion tree for some algorithm, summing the required work over all of its nodes allows us to obtain the time complexity of the algorithm.

The node representing a problem of time n is denoted by



where $f(n)$ is the work required, and each of the "a" lines lead to a problem of size n/b.

Example 12. Consider the recurrence $T_n = 3T_{n/4} + n$.
Its recursion tree is given below.



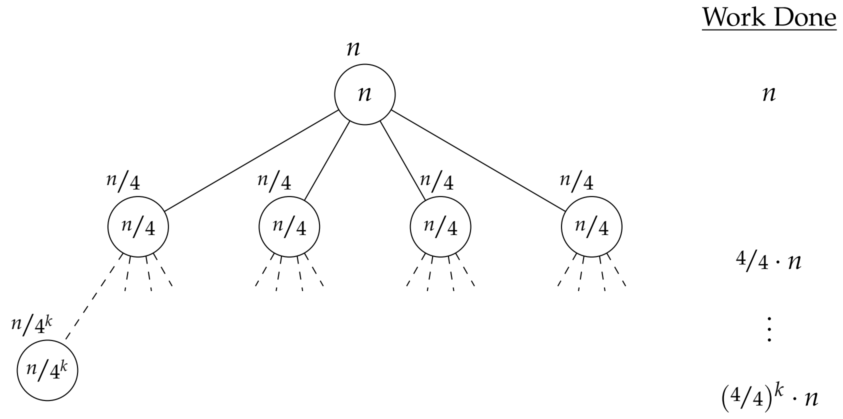
Examining the work done at each level, we see that most of the work is done at the top of the tree (early in the recursion). Observe that $k \approx \log_4(n)$ since $n/4^k \approx 1$. The total work is

$$T_n = n\left[1 + 3/4 + (3/4)^2 + \cdots + (3/4)^k\right] \leq n\sum_{i=0}^{\infty}(3/4)^i = 4n,$$

where we have used the fact that $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ for any $x \in (0,1)$.

Just as the master theorem predicts, we conclude that $T_n = O(n)$.

Example 13. Consider the recurrence $T_n = 4 \cdot T_{n/4} + n$.
Its recurrence tree is given below.

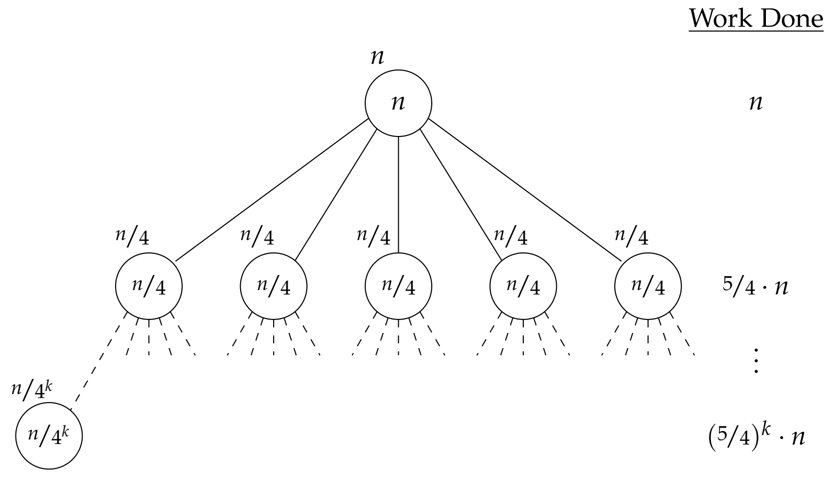


Examining the work done at each level, we see that it is distributed uniformly about each level. The total work is

$$T_n = \Theta(n \log(n)),$$

since we do n work on $k \approx \log_4(n)$ levels. Again, this is confirmed by the master theorem.

Example 14. Consider the recurrence $T_n = 5 \cdot T_{n/4} + n$.
Its recurrence tree is given below.



Examining the work done at each level, we see that most of the work is done at the bottom of the tree (late in the recursion). At the last level we do $(5/4)^k \cdot n = 5^k$ work (since $k \approx \log_4(n)$) so the work done here is $5^{\log_4(n)} = n^{\log_4(5)}$. The total work done is

$$T_n = n^{\log_4(5)} \left(1 + 4/5 + (4/5)^2 + \cdots + (4/5)^k\right) \leq n^{\log_4(5)} \sum_{i=0}^{\infty} (4/5)^i = 5n^{\log_4(5)},$$

as predicted by the master theorem.

Matrix Multiplication (Strassen's Algorithm)

Consider the problem of multiplying two $n \times n$ matrices, $A \times B = C$. A naive algorithm for computing this would be the standard matrix multiplication algorithm:

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

which takes $\Theta(n^3)$ time in the RAM model of computation. Divide-and-conquer methods allow us to improve on this. We split A and B into four $n/2 \times n/2$ sub-matrices[4] and multiply them instead. This allows us to restate our problem as

[4] This is only possible if $n = 2^k$ for some $k \in \mathbb{N}$ so we pad the matrices A and B with zeros if necessary.

$$A = \left[\quad \right] \Longrightarrow \left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array} \right]$$

$$A \times B = \left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array} \right] \times \left[\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array} \right] = \left[\begin{array}{c|c} A_1B_1 + A_2B_2 & A_1B_2 + A_2B_4 \\ \hline A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{array} \right]$$

which has its recurrence given by

$$T_n = n^2 + 8T_{n/2}.$$

We do n^2 work during positioning and $8T_{n/2}$ work doing multiplications.

Despite the above trick, applying the master theorem results in the same asymptotic bound $\Theta(n^3)$ as for the naive method. Using a cleverly defined set of operations, we can get away with performing only 7 multiplications.[5] Therefore, the new recurrence is given by

$$T_n = n^2 + 7T_{n/2}$$

which yields $T_n = \Theta(n^{\log_2(7)}) \approx \Theta(n^{2.807})$ by the master theorem.

[5] Defining the following operations

$$\begin{aligned} M_1 &= (A_1 + A_4)(B_1 + B_4) \\ M_2 &= (A_3 + A_4)B_1 \\ M_3 &= A_1(B_2 - B_4) \\ M_4 &= A_4(B_3 - B_1) \\ M_5 &= (A_1 + A_2)B_4 \\ M_6 &= (A_3 - A_1)(B_1 + B_2) \\ M_7 &= (A_2 - A_4)(B_3 + B_4) \end{aligned}$$

allows us to express the matrix C as

$$\left[\begin{array}{c|c} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ \hline M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{array} \right]$$

Remark: Many years after Strassen's discovery, faster methods saw the light. Coppersmith and Winograd designed an $O(n^{2.378})$ algorithm in 1990 and Williams an $O(n^{2.373})$ algorithm in 2014. It is known that the best one can hope for is $O(n^2 \log n)$, so the gap between lower bound and best known algorithm is still considerable.

References

T. Cormen, C. Stein, R. Riverest, C. Leiserson: *Introduction to Algorithms*, McGraw-Hill Higher Education, Second Edition, 2011.

V. Strassen: *"Gaussian Elimination is not Optimal"*, Numerische Mathematik, vol. 13, pp. 354-356, 1969.

D. Coppersmith, S. Winograd: *"Matrix multiplication via arithmetic progressions"*, Journal of Symbolic Computation, vol. 9, pp. 251-280, 1990.

V. Williams: *"Multiplying matrices in $O(n^{2.373})$ time"*, Stanford University, 2014.