

Lecture 1: September 5

*Lecturer: Micah Adler**Scribes: Paul Dickson and Gary Holness*

1.1 Administrativa

The instructor will be out of town during the week of September 17. There will be no lecture on the 17th. The lecture for the 19th will be pre-taped on the 14th during the normal class hours of 2:05pm through 3:20pm in the normal classroom CS142. You are encouraged to attend the pre-taping.

1.2 Introduction

In this course, we will discuss the design and analysis of algorithms. We will examine algorithms abstractly rather than for a particular machine. To prove theorems about these algorithms we need a model of computation as a framework within which to work. In this course, we will use the Random Access Machine (**RAM**) model.

1.2.1 RAM model

In the RAM model of computation, the CPU can access any location in memory at unit cost. It is because of this ability to randomly access any place in memory that the model is called RAM. Throughout this course we will primarily be concerned with the performance of algorithms under this model, and will measure the performance by looking at the running time of the algorithm.

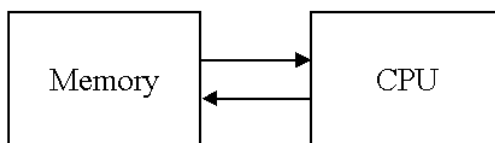


Figure 1.1: RAM model of computation

We chose the RAM model of computation because it is sufficient to describe the phenomena about which we wish to prove properties during this course. The advantages to using the RAM model are that it is:

- simple
- a fairly realistic representation

- machine independent

1.2.2 Running time

We measure the *running time* of an algorithm by tallying the total number of operations it takes. Each operation is considered to be unit-cost. The operations are:

- pairwise arithmetic operation
- access to memory
- logical operations
- comparisons (such as larger than or same)

1.2.3 Other possible models of computation

Other models of computation include the Turing Machine, Hierarchical Memory Model and Parallel Processing.

In the **Turing Machine** model memory can only be accessed sequentially [SIPSER p125]. This means that only a memory location's immediate neighbors can be accessed from the current location. While this simplification lends itself to easier theorem proving, it is unrealistic with current computers which have random access memory.

The **Hierarchical Memory Model** has a more realistic representation of memory and accounts for varying access speeds of secondary and tertiary memories such as cache and disk. This model is more exact than the RAM model but is more machine specific and far beyond the scope of this class.

Parallel Processing models take into account the aspects of parallelizing computation. This model is also beyond the scope of this class.

1.3 Asymptotic Behavior

1.3.1 Worst case running time

Input size n

$T(n)$	$n = 10$	$n = 20$	$n = 50$	$n = 100$	$n = 1000$	$n = 10^6$
$n \log n$	30 ns	90 ns	0.2 μ s	0.7 μ s	10 μ s	20 ms
n^2	100 ns	400 ns	2.5 μ s	10 μ s	1 ms	17 min
n^5	0.1 ms	3.2 ms	0.3 s	10.8 s	11.6 days	3×10^{13} yrs
2^n	1 μ s	1 ms	13 days	4×10^{13} yrs		
$n!$	4 ms	771 yrs				

Table 1.1: Time required to perform $T(n)$ operations (10^9 operations/second)

Our measure of interest, namely $T(n)$, describes the worst case running time on inputs of size n . $T(n)$ represents the running time for all inputs. Table 1.1 shows the time required to perform operations given a machine capable of executing one billion operations per second. The running times for this machine are listed for various expressions for $T(n)$ and input sizes of n ranging from ten through one billion. This illustrates the dramatic effect of increasing input size on run time. For sufficiently large input, constant factors of the input do not contribute significantly to the running time. As the input size grows, constants become insignificant when compared to the asymptotic growth rate of the function. Thus, we will not concern ourselves with constant factors.

1.3.2 Review of asymptotic notation

With asymptotic notation we will first review the upper bounds, the first of these is big O which gives an upper bound which is greater than or equal to the function. Big O is described as the following:

$$T(n) = O(f(n)) \Leftrightarrow \exists c, n_0 \text{ such that } \forall n \geq n_0, T(n) \leq cf(n)$$

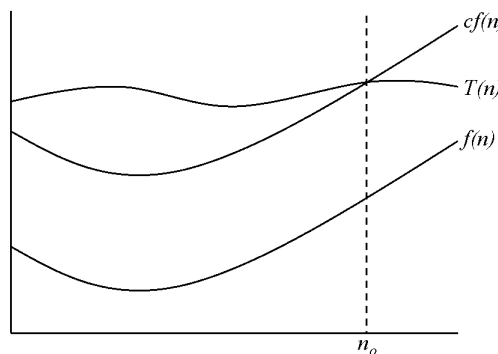


Figure 1.2: $T(n) = O(f(n))$

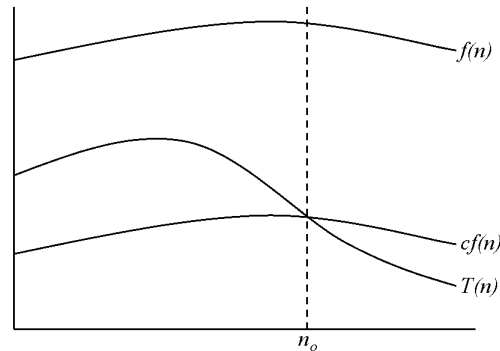
Figure 1.2 is a graphical representation of the concept showing how $T(n) \leq cf(n)$ only when $n \geq n_0$.

Figure 1.3 describes the same information but for little o notation, with $T(n) < cf(n)$ only when $n \geq n_0$. The exact description of little o is as follows:

$$T(n) = o(f(n)) \Leftrightarrow \forall c > 0, \exists n_0 \text{ such that } \forall n \geq n_0, T(n) < cf(n)$$

The second set of points to review are the lower bounds big and little Omega. Big Omega is essentially less than or equal to and is described as follows:

$$T(n) = \Omega(f(n)) \Leftrightarrow \exists c, n_0 \text{ such that } \forall n \geq n_0, T(n) \geq cf(n)$$

Figure 1.3: $T(n) = o(f(n))$

While little Omega is basically strictly less then, as described below:

$$T(n) = \omega(f(n)) \Leftrightarrow \forall c > 0, \exists n_0 \text{ such that } \forall n \geq n_0, T(n) > cf(n)$$

The final point of asymptotic notation to review is the case where bounding is equal to the function. This case is described as follows:

$$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n))$$

This information is summarized in the following table:

" \leq "	big O	$T(n) = O(f(n)) \Leftrightarrow \exists c, n_0 \text{ such that } \forall n \geq n_0, T(n) \leq cf(n)$	Figure 1.2
" $<$ "	little o	$T(n) = o(f(n)) \Leftrightarrow \forall c > 0, \exists n_0 \text{ such that } \forall n \geq n_0, T(n) < cf(n)$	Figure 1.3
" \geq "	big Omega	$T(n) = \Omega(f(n)) \Leftrightarrow \exists c, n_0 \text{ such that } \forall n \geq n_0, T(n) \geq cf(n)$	
" $>$ "	little omega	$T(n) = \omega(f(n)) \Leftrightarrow \forall c > 0, \exists n_0 \text{ such that } \forall n \geq n_0, T(n) > cf(n)$	
" $=$ "	Theta	$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n))$	

1.4 First Algorithmic Paradigm: Divide and Conquer

Divide and conquer algorithms work as the name implies by breaking the problem down into smaller and smaller subproblems until the subproblems are of a size that is trivial to solve, before recombining the solution.

1.4.1 Merge sort

Merge sort starts with an unsorted list of numbers. First divide the list into two halves (the divide step), then recursively solve the two subproblems (recurse step), finally the result of the subproblems is combined

(merge step). We can perform the merge in linear time by setting a pointer to each list and adding the elements one by one in sorted order to merged list[CLR p14].

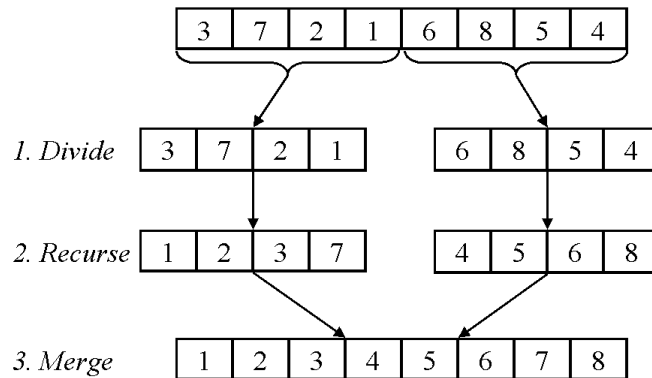


Figure 1.4: Example of merge sort

Figure 1.4 depicts a merge sort. We express the running time using a recurrence relation:

$$\begin{aligned}
 T(n) &= \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{2 subproblems}} + \underbrace{\Theta(n)}_{\text{merge}} \quad \text{for } n > 1 \\
 T(1) &= \Theta(1) \quad \text{sort list of length 1 in constant time}
 \end{aligned}$$

assuming n is a power of 2 (although the analysis also works if n is not a power of 2). We've broken the problem into two subproblems, each of size $n/2$. After recursively solving the subproblems, the merge step takes linear time in n . As we recurse, the subproblems get smaller until they reach a size of 1. At this point, the algorithm has bottomed out. The recurrence relation describes the run time for inputs of size $n > 1$. We must also express what happens at the base case, which is $T(1)$ above.

1.4.2 General formula for solving recurrence relations

In divide and conquer algorithms we divide a problem of size n into a pieces where each piece is of size n/b . After recursively solving each sub-problem, it takes n^α operations to combine the sub-problems. At each level of the recurrence, the problem size is reduced by a factor of b .

The general form for the recurrence relation for the divide and conquer algorithmic paradigm is:

$$\begin{aligned}
 T(n) &= a \cdot T\left(\frac{n}{b}\right) + \Theta(n^\alpha) \quad \text{for } n > 1 \\
 T(1) &= \Theta(1) \quad \text{for } n = 1
 \end{aligned}$$

If we let $\beta = \log_b a$, then the solution to the recurrence is dependent upon the relationship of α to β :

$$T(n) = \begin{cases} \Theta(n^\alpha) & \text{if } \alpha > \beta \\ \Theta(n^\beta) & \text{if } \alpha < \beta \\ \Theta(n^\alpha \log n) & \text{if } \alpha = \beta \end{cases} \quad (1.1)$$

In the case of merge sort, $a = b = 2$, $\beta = \log_2 2 = 1$, and $\alpha = 1 = \beta$. Since $\beta = \alpha$ we know from Equation 1.1 that the solution must be of the form $\Theta(n^\alpha \log n)$. Then substituting in the value $\alpha = 1$ into this general form we obtain $T(n) = \Theta(n \log n)$, as expected.

A recursion tree is a good representation for showing the intuition behind Equation 1.1. Assume the recursion tree shown in Figure 1.5 where each node has a possible children. The running time for each level of the tree is listed down the left hand side of the figure. The size of the subproblems for each level is listed down the right side of the table. To tally the total running time, we must account for the contribution of the subproblems at each level.

At each level we spend n^α time per subproblem for the divide and combine steps where n is the size of the subproblem. From one level to the next, the size of the input reduces $\log_b n$ many times by a factor of b until the algorithm reaches bottom. Given a level in the computation tree at depth d from the root, there are a^d subproblems. The level cost in the tree is simply the sum of the cost across subproblems. We assume that the size of the original input n is a power of b . Thus, the height of the tree is $\log_b n$.

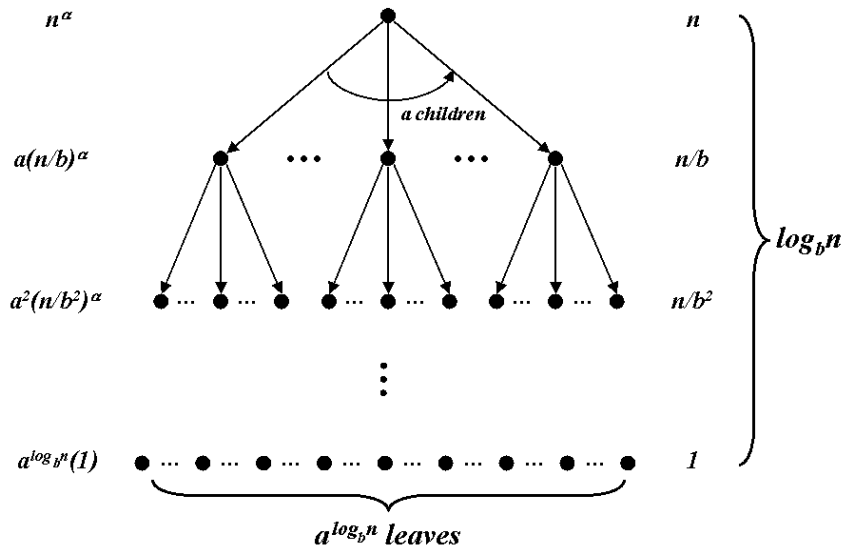


Figure 1.5: Recursion tree

By adding the running times of each level of the table we get the total running time of the tree to be:

$$T(n) = n^\alpha + a \cdot \left(\frac{n}{b}\right)^\alpha + a^2 \cdot \left(\frac{n}{b^2}\right)^\alpha + \dots + a^{\log_b n}$$

If we then let $r = \frac{a}{b^\alpha} = b^{\beta-\alpha}$, then by factoring out n^α we get:

$$T(n) = n^\alpha(1 + r + r^2 + \dots + r^{\log_b n})$$

Recall [CLR p44] that if $r \neq 1$:

$$\sum_{i=0}^k r^i = \frac{r^{k+1} - 1}{r - 1}$$

We now consider the three cases of Equation 1.1.

Case 1: for $r < 1$ ($\alpha > \beta$) we have:

$$\begin{aligned} T(n) &= n^\alpha \left(\frac{1 - r^{\log_b n + 1}}{1 - r} \right) \\ &= \Theta(n^\alpha) \end{aligned}$$

Note that this expression is less than:

$$T(n) = n^\alpha \left(\frac{1}{1 - r} \right)$$

Since r is a constant, and in asymptotic notation, we ignore constants, we get $\Theta(n^\alpha)$. With this case, the time spent at the root of the recursion tree dominates the running time.

Case 2: for $r > 1$ ($\alpha < \beta$) we have:

$$\begin{aligned} T(n) &= n^\alpha \left(\frac{r^{\log_b n + 1} - 1}{r - 1} \right) \\ &= \Theta(n^\alpha \cdot r^{\log_b n}) && \text{recall that } r = b^{\beta - \alpha} \\ &= \Theta(n^\alpha \cdot (b^{\beta - \alpha})^{\log_b n}) \\ &= \Theta(n^\alpha \cdot n^{\beta - \alpha}) \\ &= \Theta(n^\beta) \end{aligned}$$

As the terms get larger and larger, the last term dominates. This case corresponds to the leaves of the computation tree dominating the running time.

Case 3: for $r = 1$ ($\alpha = \beta$) we have

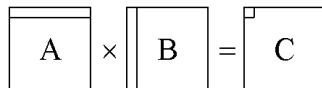
$$\begin{aligned} T(n) &= n^\alpha \overbrace{(1 + 1 + \dots + 1)}^{\log_b n \text{ times}} \\ &= \Theta(n^\alpha \log n) \end{aligned}$$

Time spent at each level of the tree is comparable, so total running time is the time at each level (n^α) multiplied by the number of levels in the tree ($\log_b n$).

1.4.3 Matrix multiplication

Input: two $n \times n$ matrices A and B , having entries from any ring (*e.g.*, the real numbers)

Output: $C = A \times B$



$$\boxed{A} \times \boxed{B} = \boxed{C}$$

Figure 1.6: Matrix multiplication

Running time of the naive matrix multiplication algorithm is $T(n) = O(n^3)$ where n is one dimension of the matrix. To improve on the running time for this matrix multiplication algorithm, lets begin by examining the multiplication of two n -bit integers.

Input: A, B : n -bit numbers
Output: $C = A \cdot B$

The naive approach to integer multiplication is $O(n^2)$. We can do better with a divide and conquer algorithm. Divide each integer into two halves of $n/2$ bits each (see Figure 1.7).

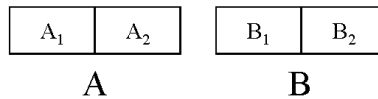


Figure 1.7: Integer multiplication

From Figure 1.7 we see that A and B are defined as:

$$\begin{aligned} A &= A_1 \cdot 2^{n/2} + A_2 \\ B &= B_1 \cdot 2^{n/2} + B_2 \end{aligned}$$

This definition of A and B means that $A \cdot B$ can be written as:

$$A \cdot B = (A_1 B_1) 2^n + (A_1 B_2 + A_2 B_1) 2^{n/2} + A_2 B_2 \quad (1.2)$$

Stay tuned next lecture for the thrilling conclusion...

References

CLR T. CORMEN and C. LEISERSON and R. RIVEST, Introduction to Algorithms, *MIT Press*, 1990.

SIPSER M. SIPSER, Introduction to the Theory of Computation, *PWS Publishing Company*, 1997.

SCRIBE2000 H. BLAU and J. NEVILLE, Scribe Notes for CS611 Advanced Algorithms Lecture 1, *UMass Amherst Computer Science*, 2000.