3.1 Asymptotic Notation

A function $f(n)$ is $\Theta(g(n))$ iff $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for some $n \geq n_0$.

This can be rewritten as

$$c_1 \leq \lim_{x \to \infty} \frac{f(n)}{g(n)} \leq c_2$$

A function $f(n)$ is $\Theta(g(n))$ iff $f(n) = \Omega(g(n)) \wedge f(n) = \mathcal{O}(g(n))$.

A function $f(n)$ is $o(g(n))$ iff

$$\lim_{x \to \infty} \frac{f(n)}{g(n)} = 0$$

A function $f(n)$ is $\omega(g(n))$ iff

$$\lim_{x \to \infty} \frac{g(n)}{f(n)} = 0$$

In general, we can state the following about asymptotics, colloquially:

$f(n) = \mathcal{O}(g(n)) \implies f(n) \leq g(n)$

$f(n) = \Omega(g(n)) \implies f(n) \geq g(n)$

$f(n) = \Theta(g(n)) \implies f(n) = g(n)$

$f(n) = o(g(n)) \implies f(n) < g(n)$

$f(n) = \omega(g(n)) \implies f(n) > g(n)$

Asymptotics also maintain a few properties similar to equations, such as transitivity, reflexivity, and symmetry.

Transitivity:

If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$.

If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.

If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.

If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$.

If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$.

Reflexivity:

$f(n) = \mathcal{O}(f(n))$

$f(n) = \Omega(f(n))$

$f(n) = \Theta(f(n))$

Symmetry:

If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$.

Transpose Symmetry:

If $f(n) = \mathcal{O}(g(n))$, then $g(n) = \Omega(f(n))$.

If $f(n) = \Omega(g(n))$, then $g(n) = \mathcal{O}(f(n))$.

Performing arithmetic on asymptotics is similar to equations as well.

Addition:

$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max(f(n) + g(n)))$

Proof: Let $F(n) = \mathcal{O}(f(n))$, and $G(n) = \mathcal{O}(g(n))$. Then $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = F(n) + G(n)$, and $F(n) + G(n) <= f(n) + g(n)$. Thus, $F(n) + G(n) = \mathcal{O}(f(n) + g(n))$.

This proof is similar for both $\Omega(f(n)) + \Omega(g(n))$ and $\Theta(f(n)) + \Theta(g(n))$. ■

$\Omega(f(n)) + \Omega(g(n)) = \Omega(\max(f(n) + g(n)))$

$\Theta(f(n)) + \Theta(g(n)) = \Theta(\max(f(n) + g(n)))$

Multiplying by Constants: Since constants are hidden in asymptotic notation, we can say the following:

$\mathcal{O}(c * f(n)) = \mathcal{O}(f(n))$

$\Omega(c * f(n)) = \Omega(f(n))$

$\Theta(c * f(n)) = \Theta(f(n))$

Multiplying Functions:

$\mathcal{O}(f(n)) * \mathcal{O}(g(n)) = \mathcal{O}((f(n) * g(n)))$

$\Omega(f(n)) * \Omega(g(n)) = \Omega((f(n) * g(n)))$

$\Theta(f(n)) * \Theta(g(n)) = \Theta((f(n) * g(n)))$

Dividing Functions:

$\frac{\mathcal{O}(f(n))}{\Theta(g(n))} = \mathcal{O}\left(\frac{f(n)}{g(n)}\right)$

Proof: Let $F(n) = \mathcal{O}(f(n))$ and $G(n) = \Theta(g(n))$. It can be said that $\frac{F(n)}{G(n)} \leq \frac{f(n)}{g(n)}$ because by definition of Big Oh, $F(n) \leq c * f(n)$, and consequently, $\frac{F(n)}{G(n)} = \mathcal{O}\left(\frac{f(n)}{g(n)}\right)$.

This proof is similar for other combinations. ■

$\frac{\Omega(f(n))}{\Theta(g(n))} = \Omega\left(\frac{f(n)}{g(n)}\right)$

$\frac{\Theta(f(n))}{\Theta(g(n))} = \Theta\left(\frac{f(n)}{g(n)}\right)$

$\frac{\mathcal{O}(f(n))}{\Omega(g(n))} = \mathcal{O}\left(\frac{f(n)}{g(n)}\right)$

$\frac{\Theta(f(n))}{\Omega(g(n))} = \mathcal{O}\left(\frac{f(n)}{g(n)}\right)$

$\frac{\Omega(f(n))}{\mathcal{O}(g(n))} = \Omega\left(\frac{f(n)}{g(n)}\right)$

$\frac{\Theta(f(n))}{\mathcal{O}(g(n))} = \Omega\left(\frac{f(n)}{g(n)}\right)$

An application of dividing functions can be seen in parallel algorithms.

Let work, the optimal running time of one processor, be defined as $T_1(n)$, and let span, the theoretical running time of infinite processors, be defined as $T_\infty(n)$. Then parallelism is defined by $p = \frac{T_1(n)}{T_\infty(n)}$, which is an upper bound on speedup, s, such that $s \leq p$, i.e. $s = \mathcal{O}(p)$.

3.2 Divide and Conquer

Trominoes are a L-shaped tile made of three squares. A puzzle involving trominoes is to cover a 2^n by 2^n board with trominoes, for some $n \in \mathbb{Z}$, given that the board has a missing square.

We can solve this puzzle by placing a tromino in the center with the missing square facing the quadrant with the missing tile, and looking at the subproblem of finding a solution to the tromino problem of a 2^{n-1} by 2^{n-1} board. We continue doing this until we hit a base case of a board of size 2 by 2, and we can solve this by placing a tromino in the missing slot because there is a square missing. We merge these solutions together to get a board filled with trominoes.

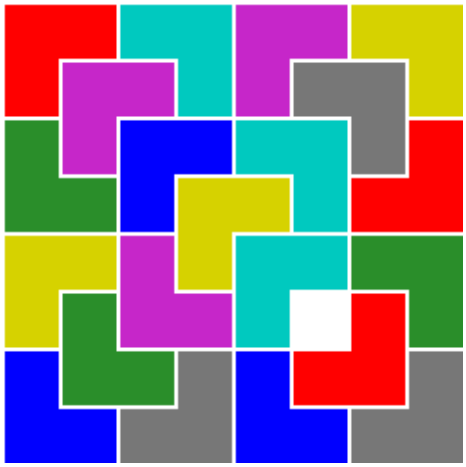

Figure 3.2.1: A picture of a 2^n by 2^n board with a missing square filled with trominoes.

In general, this is the process for divide and conquer. First, we have an original problem and divide it into smaller subproblems such that they are identical to the original problem. Secondly, we conquer the smallest subproblems. Finally, we merge the solutions to get an answer for the original problem.

An example of this is merge sort, which takes the form of a piecewise function:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) & n \geq 1 \\ \Theta(1) & n = 1 \end{cases}$$

This recurrence equates to $\Theta(nlgn)$.

3.3 Integer Multiplication

When multiplying two n-bit integers, we can perform a naive algorithm by long multiplication. This takes $\Theta(n^2)$ time, which has the same time complexity of the divide and conquer technique for multiplying integers.

Let x be an n-bit integer, such that it can be separated into two sections, x_L and x_R. We can say that $x = 2^{n/2}x_L + x_R$. Let y be another n-bit integer, such that it can be separated into two sections, y_L and y_R. We can say that $y = 2^{n/2}y_L + y_R$.

This is similar to representing numbers in base 10 as two sums. For example, 583926 can be written as $(583) * 10^3 + 926$.

Going back to the numbers x and y, we can say that

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R)$$

$$xy = 2^n x_L y_L + (2^{n/2})(x_R y_L + x_L y_R) + x_R y_R$$

This performs two bit shifts, three additions, and 4 $n/2$ bit multiplications. This total work takes $\Theta(n)$ and thus, the recurrence takes the form of a piecewise function:

$$T(n) = \begin{cases} 4T(\frac{n}{2}) + \Theta(n) & n \geq 1 \\ \Theta(1) & n = 1 \end{cases}$$

This recurrence equates to $\Theta(n^2)$.

A more complex algorithm is named Karatsuba's algorithm which has a time complexity of $\Theta(n^{1.59})$.

Proof: Karatsuba converted the $(x_R y_L + x_L y_R)$ in the divide and conquer algorithm to $(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$. This is true because

$$(x_L + x_R)(y_L + y_R) = x_L y_L + x_R y_L + x_L y_R + x_R y_R$$

$$(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R = x_R y_L + x_L y_R$$

This brings the number of $n/2$ bit multiplications to 3, and brings the following piecewise function and recurrence:

$$T(n) = \begin{cases} 3T(\frac{n}{2}) + \Theta(n) & n \geq 1 \\ \Theta(1) & n = 1 \end{cases}$$

This is $\Theta\left(n^{\log_2 3}\right) = \Theta\left(n^{1.59}\right)$

■

There are multiple algorithms for integer multiplication, the fastest being $nlogn2^{\mathcal{O}(logn)}$

3.4 Matrix Multiplication

Matrix multiplication can be done naively in $\Theta\left(n^3\right)$ by iteratively multiplying rows by columns for each element.

$$\begin{bmatrix} xy_{11} & xy_{12} & xy_{13} & \dots & xy_{1n} \\ xy_{21} & xy_{22} & xy_{23} & \dots & xy_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ xy_{d1} & xy_{d2} & xy_{d3} & \dots & xy_{dn} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & x_{d3} & \dots & x_{dn} \end{bmatrix} * \begin{bmatrix} y_{11} & y_{12} & y_{13} & \dots & y_{1n} \\ y_{21} & y_{22} & y_{23} & \dots & y_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{d1} & y_{d2} & y_{d3} & \dots & y_{dn} \end{bmatrix}$$

such that xy_{ij} for some $0 \leq i, j \leq n$ and $i, j \in \mathbb{Z}$ is equal to $X_m Y_n$ where X_m represents the mth row, and Y_n represents the nth column.

Each cell takes $\Theta(n)$ time to compute, for $\Theta\left(n^2\right)$ cells for a total of $\Theta\left(n^3\right)$ time.

Pseudocode for this implementation for matrix multiplcation can be seen below:

MATRIXMULTIPLICATION(z, x, y)
(Inputs are three n by n matrices, where z is the output matrix, i.e. the product of x and y)

1. ***if*** $x =$ null or $y =$ null ***then return*** {*either of the matrices are null*}
2. ***else*** ***for*** $i \leq n$ ***do*** {i *represents the row*}
   for $j \leq n$ ***do*** {j *represents the column*}
   for $k \leq n$ ***do*** {*operate on the kth index*}
   $z[i][j] = x[i][k] + y[k][j]$ {*put output into z, the output matrix*}

MATRIXMULTIPLICATION ENDS

This is inefficient for caches of size $B < n$ because in either row-ordered orientation in main memory or column-ordered orientation in main memory requires the cache to be refreshed n times. Calling the cache this many times slows down the algorithm significantly, and there are better algorithms for matrix multiplication.