Algorithms and Data Structures

Christian J. Rudder

October 2024

Contents

This page is left intentionally blank.

— 1 —

Asymptotic Notation

1.1 Asymptotic Notation

Asymptotic analysis is a method for describing the limiting behavior of functions as inputs grow infinitely.

Definition 1.1: Asymptotic

Let $f(n)$ and $g(n)$ be two functions. As n grows, if $f(n)$ grows closer to $g(n)$ never reaching, we say that "$f(n)$ is **asymptotic** to $g(n)$."

We call the point where $f(n)$ starts behaving similarly to $g(n)$ the **threshold** n_0. After this point n_0, $f(n)$ follows the same general path as $g(n)$.

Definition 1.2: Big-O: (Upper Bound)

Let f and g be functions. $f(n)$ our function of interest, and $g(n)$ our function of comparison.

Then we say $f(n) = O(g(n))$, "$f(n)$ **is big-O of** $g(n)$," if $f(n)$ grows no faster than $g(n)$, up to a constant factor. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq f(n) \leq c \cdot g(n)$$

Represented as the ratio $\dfrac{f(n)}{g(n)} \leq c$ for all $n \geq n_0$. Analytically we write,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Meaning, as we chase infinity, our numerator grows slower than the denominator, bounded, never reaching infinity.

Examples:

(i.) $3n^2 + 2n + 1 = O(n^2)$

(ii.) $n^{100} = O(2^n)$

(iii.) $\log n = O(\sqrt{n})$

Proof 1.1: $\log n = O(\sqrt{n})$

We setup our ratio:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}}$$

Since $\log n$ and $\sqrt{n}$ grow infinitely without bound, they are of indeterminate form $\frac{\infty}{\infty}$. We apply L'Hopital's Rule, which states that taking derivatives of the numerator and denominator will yield an evaluateable limit:

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\frac{d}{dn}\log n}{\frac{d}{dn}\sqrt{n}}$$

Yielding derivatives, $\log n = \frac{1}{n}$ and $\sqrt{n} = \frac{1}{2\sqrt{n}}$. We substitute these back into our limit:

$$\lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = \lim_{n\to\infty} \frac{2\sqrt{n}}{n} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

Our limit approaches 0, as we have a constant factor in the numerator, and a growing denominator. Thus, $\log n = O(\sqrt{n})$, as $0 < \infty$. ■

Definition 1.3: Big-Ω: (Lower Bound)

The symbol Ω reads "Omega." Let f and g be functions. Then $f(n) = \Omega(g(n))$ if $f(n)$ grows no slower than $g(n)$, up to a constant factor. I.e., lower bounded by $g(n)$. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq c \cdot g(n) \leq f(n)$$

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

Meaning, as we chase infinity, our numerator grows faster than the denominator, approaching 0 asymptotically.

Examples: $n! = \Omega(2^n)$; $\frac{n}{100} = \Omega(n)$; $n^{3/2} = \Omega(\sqrt{n})$; $\sqrt{n} = \Omega(\log n)$

Definition 1.4: Big Θ: (Tight Bound)

The symbol Θ reads "Theta." Let f and g be functions. Then $f(n) = \Theta(g(n))$ if $f(n)$ grows at the same rate as $g(n)$, up to a constant factor. I.e., $f(n)$ is both upper and lower bounded by $g(n)$. Let n_0 be our asymptotic threshold, and $c_1 > 0, c_2 > 0$ be some constants. Then, for all $n \geq n_0$,

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

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

Meaning, as we chase infinity, our numerator grows at the same rate as the denominator.

Examples: $n^2 = \Theta(n^2)$; $2n^3 + 2n = \Theta(n^3)$; $\log n + \sqrt{n} = \Theta(\sqrt{n})$.

Definition 1.5: Little o: (Strict Upper Bound)

The symbol o reads "little-o." Let f and g be functions. Then $f(n) = o(g(n))$ if $f(n)$ grows strictly slower than $g(n)$, meaning $f(n)$ becomes insignificant compared to $g(n)$ as n grows large. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq f(n) < c \cdot g(n)$$

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

Meaning, as we chase infinity, the ratio of $f(n)$ to $g(n)$ shrinks to zero.

Examples: $n = o(n^2)$; $\log n = o(n)$; $n^{0.5} = o(n)$.

Definition 1.6: Little ω: (Strict Lower Bound)

The symbol ω reads "little-omega." Let f and g be functions. Then $f(n) = \omega(g(n))$ if $f(n)$ grows strictly faster than $g(n)$, meaning $g(n)$ becomes insignificant compared to $f(n)$ as n grows large. Let n_0 be our asymptotic threshold. Then, for all $n \geq n_0$,

$$0 \leq c \cdot g(n) < f(n)$$

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

Meaning, as we chase infinity, the ratio of $g(n)$ to $f(n)$ shrinks to zero.

Examples: $n^2 = \omega(n)$; $n = \omega(\log n)$.

Definition 1.7: Asymptotic Equality ($\sim$)

The symbol $\sim$ reads "asymptotic equality." Let f and g be functions. Then $f(n) \sim g(n)$ if, as $n \to \infty$, the ratio of $f(n)$ to $g(n)$ approaches 1. I.e., the two functions grow at the same rate asymptotically. Formally,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

Meaning, as n grows large, the two functions become approximately equal.

Examples: $n + 100 \sim n$, $\log(n^2) \sim 2\log n$.

Tip: To review:

- **Big-O:** $f(n) < g(n)$ (Upper Bound); $f(n)$ grows no faster than $g(n)$.
- **Big-Ω:** $f(n) > g(n)$ (Lower Bound); $f(n)$ grows no slower than $g(n)$.
- **Big-Θ:** $f(n) = g(n)$ (Tight Bound); $f(n)$ grows at the same rate as $g(n)$.
- **Little-o:** $f(n) < g(n)$ (Strict Upper Bound); $f(n)$ grows strictly slower than $g(n)$.
- **Little-ω:** $f(n) > g(n)$ (Strict Lower Bound); $f(n)$ grows strictly faster than $g(n)$.
- **Asymptotic Equality:** $f(n) \sim g(n)$; $f(n)$ grows at the same rate as $g(n)$.

Theorem 1.1: Types of Asymptotic Behavior

The following are common relationships between different types of functions and their asymptotic growth rates:

- **Polynomials.** Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ with $a_d > 0$. Then, $f(n)$ is $\Theta(n^d)$.
  E.e., $3n^2 + 2n + 1$ is $\Theta(n^2)$.
- **Logarithms.** $\Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a, b > 0$. That is, logarithmic functions in different bases have the same growth rate.
  E.g., $\log_2 n$ is $\Theta(\log_3 n)$.
- **Logarithms and Polynomials.** For every $d > 0$, $\log n$ is $O(n^d)$. This indicates that logarithms grow slower than any polynomial.
  E.g., $\log n$ is $O(n^2)$.
- **Exponentials and Polynomials.** For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$. This means that exponentials grow faster than any polynomial.
  E.e., n^2 is $O(2^n)$.

1.2 Evaluating Algorithms

When analyzing algorithms, we are interested in two primary factors: time and space complexity.

Definition 2.1: Time Complexity

The **time complexity** of an algorithm is the amount of time it takes to run as a function of the input size. We use asymptotic notation to describe the time complexity of an algorithm.

Definition 2.2: Space Complexity

The **space complexity** of an algorithm is the amount of memory it uses to store inputs and subsequent variables during the algorithm's execution. We use asymptotic notation to describe the space complexity of an algorithm.

Below is an example of a function and its time and space complexity analysis.

Function 2.1: Arithmetic Series - `Fun1(`A`)`

Computes a result based on a length-n array of integers:

Input: A length-n array of integers.
Output: An integer p computed from the array elements.

```
1 Function Fun1(A):
2 |  p ← 0;
3 |  for i ← 1 to n − 1 do
4 |  |  for j ← i + 1 to n do
5 |  |  |  p ← p + A[i] · A[j];
```

Time Complexity: For $f(n) :=$ `Fun1(`A`)`, $f(n) = \frac{n^2}{2} = O(n^2)$. This is because the function has a nested loop structure, where the inner for-loop runs $n - i$ times, and the outer for-loop runs $n - 1$ times. Thus, the total number of iterations is $\sum_{i=1}^{n-1} n - i = \frac{n^2}{2}$.

Space Complexity: We yield $O(n)$ for storing an array of length n. The variable p is $O(1)$ (constant), as it is a single integer. Hence, $f(n) = n + 1 = O(n)$.

Additional Example: Let $f(n, m) = n^2m + m^3 + nm^3$. Then, $f(n, m) = O(n^2m + m^3)$. This is because both n and m must be accounted for. Our largest n term is n^2m, and our largest m term is m^3 both dominate the expression. Thus, $f(n, m) = O(n^2m + m^3)$.

— 2 —

Computers and Number Based Systems

2.1 Computers & Number Base Systems

Definition 1.1: Turing Machine

A **Turing Machine** is a theoretical computational model used to describe the capabilities of a general-purpose **computer**. It consists of an infinite tape (memory) and a read/write head processing symbols on the tape, one at a time, according to a set of predefined rules. The machine moves left or right, reading or writing symbols, and changing states based on what it reads.

The machine **halts** once it reaches a final state or continues indefinitely. Serving as a flexible, **higher-order function** (a function which receives functions).

Definition 1.2: Von Neumann Architecture

Modern computers operate on a model known as the **Von Neumann architecture**, which consists of three primary components:

1. **Memory**: Stores data and instructions as sequences of bits.
2. **Arithmetic and Logic Unit (ALU)**: Executes operations such as addition, subtraction, multiplication, and division on numbers stored in memory.
3. **Control Unit**: Directs the execution of instructions and manages the flow of data between memory and the ALU.

Where numbers are stored in memory cells, each cell holding an integer value represented in a fixed base, typically $B = 2$, meaning **binary**. Where each digit is less than the base B. We represent integers in memory as:

$$a = \sum_{i=0}^{k-1} a_i B^i$$

where a_i represents the individual digits, and B is the base. For large integers, computations may require manipulating several memory cells to store the full number.

Example: Consider the integer $a = 13$, and let us represent it in base $B = 2$ (binary). We can express this number as a sum of powers of 2, corresponding to the binary representation of 13:

$$a = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13$$

In binary, this is represented as the sequence of digits: $a = (1101)_2$. Here, the coefficients $a_3 = 1$, $a_2 = 1$, $a_1 = 0$, and $a_0 = 1$ correspond to the binary digits of 13.
Similarly, if we want to represent $a = 45$ in base $B = 10$ **(decimal)**, we write:

$$a = 4 \cdot 10^1 + 5 \cdot 10^0 = 40 + 5 = 45$$

In this case, the coefficients $a_1 = 4$ and $a_0 = 5$ correspond to the decimal digits of 45.

Definition 1.3: Hexadecimal

Hexadecimal base $B = 16$, using digits 0-9 and the letters A-F, where $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, and $F = 15$. Hexadecimal is commonly used in computing due to its compact representation of binary data. For example, a **byte** (8 bits) can be represented as two hexadecimal digits, simplifying the display of binary data.

Theorem 1.1: Base $2 \leftrightarrow 16$ Conversion

Let bases $B := 2$ (binary) and $H := 16$ (hexadecimal). At a high-level:

Binary to Hexadecimal:

1. Group B digits in sets of 4, right to left. **Pad** leftmost group with 0's if necessary for a full group.
2. Compute each group, replacing the result with their H digit.
3. Finally, combine each H group.

Hexadecimal to Binary:

1. Convert each H digit into a 4 bit B group.
2. Finally, combine all B groups.

Additionally we may also trim any leading 0's.

Example:

- Binary to Hexadecimal:

$$101101111010_2 \quad \Rightarrow \quad \text{Group as ([1011] [0111] [1010)} \quad \Rightarrow \quad B7A_{16}$$

- Hexadecimal to Binary:

$$3F5_{16} \quad \Rightarrow \quad [0011]\ [1111]\ [0101]_2 \Rightarrow \quad 1111110101_2$$

The following definition is for completeness: applications of such a base are currently uncommon.

Definition 1.4: Unary

Unary, base $B = 1$. A system where each number is represented by a sequence of B symbols. Where the number n is represented by n symbols. Often used in theoretical computer science to prove the existence of computable functions.

Example: The number 5 in unary is represented by 5 symbols: $5 = \texttt{IIIII}$ or $2 = \texttt{II}$. There is no concept of 0, the absence of symbols represents 0.

Definition 1.5: Most & Least Significant Bit

In a binary number, the **most significant bit (MSB)** is the leftmost bit. The **least significant bit (LSB)** is the rightmost bit.

Example: Consider this byte (8 bits), $[1111\ 1110]_2$, the MSB = 1 and the LSB = 0.

Theorem 1.2: Adding Binary

We may use the add and carry method alike decimal addition: **Binary Addition Rules:**

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ (add 1 to the next digit (left))

We call the last step, **carry**, as we carry our overflow to the next digit.

Example: Adding $0010\ 0011\ 0100_2$ and 0100_2:

$$\begin{array}{r} \overset{1}{} \\ 1\,1000\,0100 \\ +\,0\,0001\,0100 \\ \hline 1\,1001\,1000 \end{array}$$

Where $[1\ 1000\ 0100]_2 + [0\ 0001\ 0100]_2 = [1\ 1001\ 1000]_2$.

Theorem 1.3: Signed Binary Numbers - Two's Complement

In a **two's complement system**, an n-bit signed (positive or negative) binary number can represent values in the range $[-2^{n-1}, 2^{n-1} - 1]$. Then by most significant bit (MSB):

- If MSB is 0, the number is positive;
- If MSB is 1, the number is negative.

Conversion to Two's Complement :

1. Take an unsigned binary number and invert all bits, turning 0's to 1's and 1's to 0's.
2. Finally add 1 to the least significant bit.

Example: Converting -5 into a 4-bit two's complement:

$$\begin{aligned} 5 \to 0101 \quad & \text{(binary for 5)} \\ 1010 \quad & \text{(inverted)} \\ 1011 \quad & \text{(add 1)} \end{aligned}$$

Thus, -5 is represented as 1011 in 4-bits under two's complement.

2.2 Computing Large Numbers

In this section we discuss algorithms for computing large numbers, but first we define algorithmically, addition, subtraction, multiplication, division, and modula.

Definition 2.1: Wordsize

Our machine has a fixed **wordsize**, which is how much each memory cell can hold. Systems like 32-bit or 64-bit can hold 2^{32} (≈ 4.3 billion) or 2^{64} (≈ 18.4 quintillion) bits respectively.

We say the ALU can performs arithmetic operations at $O(1)$ time, within wordsize. Operations beyond this size we deem **large numbers**.

The game we play in the following algorithms is to compute large integers without exceeding wordsize. Moving forward, we assume our machine is a typical 64-bit system.

Function 2.1: Length of digits - $\|a\|$

We will use the notation $\|a\|$ to denote the number of digits in the integer a. For example, $\|123\| = 3$ and $\|0\| = 1$.

Definition 2.2: Computer Integer Division

Our ALU only returns the quotient after division. We denote the quotient as $\lfloor a/b \rfloor : a, b \in \mathbb{Z}$.

Our first hurdle is long division as , which will set up long addition and subtraction for success.
Scenario - Grade School Long Division: Goes as follows, take $\frac{a}{b}$. Find how times b fits into a evenly, q times. Then $a - bq$ is our remainder r.
Examples: let $a = \{12, 5, 17, 40, 89\}$, $b = \{4, 2, 3, 9, 10\}$ respectively, and base $B = 10$,

$$\text{(1.)}\quad \begin{array}{r} 3 \\ 4\overline{)12} \\ \underline{12} \\ 0 \end{array} \quad \text{(2.)}\quad \begin{array}{r} 2 \\ 2\overline{)5} \\ \underline{4} \\ 1 \end{array} \quad \text{(3.)}\quad \begin{array}{r} 5 \\ 3\overline{)17} \\ \underline{15} \\ 2 \end{array} \quad \text{(4.)}\quad \begin{array}{r} 4 \\ 9\overline{)40} \\ \underline{36} \\ 4 \end{array} \quad \text{(5.)}\quad \begin{array}{r} 8 \\ 10\overline{)89} \\ \underline{80} \\ 9 \end{array}$$

Take (3.), $a = 17$, $b = 3$: 3 fits into 17 five times, which is 15. 17 take away 15 is 2, our remainder. We create an algorithm to compute this process.

Key Observation: Consider the following powers of 2 of form $x = 2^n + s$, where $x, n, s \in \mathbb{Z}$:

$$\begin{aligned} 3 &= 2 + 1 = 0000\ 0011_2 \quad (1) \\ 6 &= 4 + 2 = 0000\ 0110_2 \quad (2) \\ 12 &= 8 + 4 = 0000\ 1100_2 \quad (3) \\ 24 &= 16 + 8 = 0001\ 1000_2 \quad (4) \\ 48 &= 32 + 16 = 0011\ 0000_2 \quad (5) \\ 96 &= 64 + 32 = 0110\ 0000_2 \quad (6) \\ 192 &= 128 + 64 = 1100\ 0000_2 \quad (7) \end{aligned}$$

Notice that as we increase the power of 2, the number of bits shift left towards a higher-order bit. Now, instead of calculating powers of 2, we shift bits left or right, to yield instantaneous results.

Theorem 2.1: Binary Bit Shifting (Powers of 2)

Let x be a binary unsigned integer. Where "$\ll$" and "$\gg$" are left and right bit shifts:
Left Shift by k bits: $x \ll k := x \cdot 2^k$
Right Shift by k bits: $x \gg k = \lfloor x/2^k \rfloor$
Remainder: bits pushed out after right shift(s).

Example: Observe, $16 = 10000$ (4 zeros), $8 = 1000$ (3 zeros), we shift by 4 and 3 respectively:

- Instead of $3 \cdot 16$ in base 10, we can $3 \ll 4 = 48$, as $3 \cdot 2^4 = 48$.
- Conversely, Instead of $48/16$ in base 10, $48 \gg 4 = 3$, as $\lfloor 48/2^4 \rfloor = 3$.
- Catching the remainder: say we have $37/8$ base 10, then,

$$37 = 100101_2 \quad \text{and} \quad 8 = 1000_2 \text{ then } 37 \gg 3 = 4 \text{ remainder } 5,$$

as $[100101] \gg 3 = [000100]101$, where 101_2 is our remainder 5_{10}.

Function 2.2: Division with Remainder in Binary (Outline) - *QuoRem()*

For binary integers, let dividend $a = (a_{k-1} \cdots a_0)_2$ and divisor $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned, with $k \geq 1$, $\ell \geq 1$, ensuring $0 \leq b \leq a$, and $b_{\ell-1} \neq 0$, ensuring $b > 0$.

We compute q and r such that, $a = bq + r$ and $0 \leq r < b$. Assume $k \geq \ell$; otherwise, $a < b$. We set $q \leftarrow 0$ and $r \leftarrow a$. Then quotient $q = (q_{m-1} \cdots q_0)_2$ where $m := k - \ell + 1$.

Input: a, b (binary integers)
Output: q, r (quotient and remainder in binary)

```
1 Function QuoRem(a, b):
2     r ← a;
3     q ← {0_{m-1} ··· 0};
4     for i ← ||a|| − ||b|| − 1 down to 0 do
5         q_i ← ⌊ r / (b ≪ i) ⌋;
6         r ← r − (q_i · (b ≪ i));
```

Time Complexity: $O(\|a\|(\|a\| - \|b\|))$. In short, line 5 we preform division on $\|a\|$ bits of decreasing size. Though **not totally necessary**, For more detail visit https://shoup.net/ntb/ntb-v2.pdf on page 60. General n cases can be found in Theorem (2.2).

Example Let $a = 47_{10} = 101111_2$ and $b = 5_{10} = 101_2$, we run *QuoRem(a, b)*. We summarize the above example as, "How many times does 101_2 fit into 101111_2?"

1. Does $5 \ll 3$ fit into 101000_2? It fits! $q = 1000_2$, $r = 101111_2 - 101000_2$.
2. Does $5 \ll 2$ fit into 0111_2? no fits! $q = 1000_2$, $r = 0111_2$.
3. Does $5 \ll 1$ fit into 0111_2? no fits! $q = 1000_2$, $r = 0111_2$.
4. Does $5 \ll 0$ fit into 0111_2? It fits! $q = 1001_2$, $r = 0111_2 - 0101$.
5. Return $q = 1001_2 = 9_{10}$, $r = 0010 = 2_{10}$

Long addition: We craft an algorithm for grade school long addition, which goes as follows:

$$\begin{array}{r} \overset{1}{2}5\,\overset{1}{3}08 \\ +\,39\,406 \\ \hline 64\,714 \end{array}$$

Where adding, $25,308 + 39,406 = 64,714$. We create an algorithm to compute this in the following function.

Here the function *QuoRem* (2.2) to return (quotient, remainder) preforms in $O(1)$ as bits are small enough for word size.

Function 2.3: Addition of Binary Integers - *Add()*

Let $a = (a_{k-1} \cdots a_0)_2$ and $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned binary integers, where $k \geq \ell \geq 1$. We compute $c := a + b$ where the result $c = (c_k c_{k-1} \cdots c_0)_2$ is of length $k + 1$, assuming $k \geq \ell$. If $k < \ell$, swap a and b. This algorithm computes the binary representation of $a + b$.

Input: a, b (binary integers)
Output: $c = (c_k \cdots c_0)_2$ (sum of $a + b$)

```
1  Function Add(a, b):
2      carry ← 0;
3      for i ← 0 to ℓ − 1 do
4          tmp ← a_i + b_i + carry;
5          (carry, c_i) ← QuoRem(tmp, 2);
6      for i ← ℓ to k − 1 do
7          tmp ← a_i + carry;
8          (carry, c_i) ← QuoRem(tmp, 2);
9      c_k ← carry;
10     return c = (c_k ··· c_0)_2;
```

Note: $0 \leq carry \leq 1$ and $0 \leq tmp \leq 3$.
Time Complexity: $O(\max(\|a\|, \|b\|))$, as we iterate at most the length of the largest input.
Space Complexity: $O(\|a\| + \|b\|)$, though $c = k + 1$, constants are negligible as $k, \ell \to \infty$.

For subtracting, $5,308 - 3,406 = 1,904$, where we borrow 10 from the 5 to make 13:

$$\begin{array}{r} {\not{5}}^{4}\,{}^{10}3\,08 \\ -\,3\;406 \\ \hline 1\;904 \end{array}$$

Function 2.4: Subtraction of Binary Integers - *Subtract()*

Let $a = (a_{k-1} \cdots a_0)_2$ and $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned binary integers, where $k \geq \ell \geq 1$ and $a \geq b$. We compute $c := a - b$ where the result $c = (c_{k-1} \cdots c_0)_2$ is of length k, assuming $a \geq b$. If $a < b$, swap a and b and set a negative flag to indicate the result is negative. This algorithm computes the binary representation of $a - b$.

Input: a, b (binary integers)
Output: $c = (c_{k-1} \cdots c_0)_2$ (difference of $a - b$)

```
1  Function Subtract(a, b):
2      borrow ← 0;
3      for i ← 0 to ℓ − 1 do
4          tmp ← a_i − b_i − borrow;
5          if tmp < 0 then
6              borrow ← 1;
7              c_i ← tmp + 2;
8          else
9              borrow ← 0;
10             c_i ← tmp;
11     for i ← ℓ to k − 1 do
12         tmp ← a_i − borrow;
13         if tmp < 0 then
14             borrow ← 1;
15             c_i ← tmp + 2;
16         else
17             borrow ← 0;
18             c_i ← tmp;
19     return c = (c_{k−1} ··· c_0)_2;
```

Note: $0 \leq borrow \leq 1$. Subtraction may produce a borrow when $a_i < b_i$.
Time Complexity: $O(\max(\|a\|, \|b\|))$, iterating at most the length of the largest input.
Space Complexity: $O(\|a\| + \|b\|)$, as the length of c is at most k, with constants negligible as $k, \ell \to \infty$.

For multiplication, $24 \cdot 16 = 384$:

$$\begin{array}{r} {\scriptstyle 2} \\ 24 \\ \times\ 16 \\ \hline 144 \\ +240 \\ \hline 384 \end{array}$$

Where $6 \cdot 4 = 24$, we write the 4 and carry the 2. Then $6 \cdot 2 = 12$ plus the carried 2 is 14. Then we multiply the next digit, 1, we add a 0 below our 144, and repeat the process. Every new 10s place we add a 0. Then we add our two products to get 384.

We create an algorithm to compute this process in the following function:

Function 2.5: Multiplication of Base-B Integers - *Mul()*

Let $a = (a_{k-1} \cdots a_0)_B$ and $b = (b_{\ell-1} \cdots b_0)_B$ be unsigned integers, where $k \geq 1$ and $\ell \geq 1$. The product $c := a \cdot b$ is of the form $(c_{k+\ell-1} \cdots c_0)_B$, and may be computed in time $O(k\ell)$ as follows:

Input: a, b (base-B integers)
Output: $c = (c_{k+\ell-1} \cdots c_0)_B$ (product of $a \cdot b$)

```
1  Function Mul(a, b):
2      for i ← 0 to k + ℓ − 1 do
3          c_i ← 0;
4      for i ← 0 to k − 1 do
5          carry ← 0;
6          for j ← 0 to ℓ − 1 do
7              tmp ← a_i · b_j + c_{i+j} + carry;
8              (carry, c_{i+j}) ← QuoRem(tmp, B);
9          c_{i+ℓ} ← carry;
10     return c = (c_{k+ℓ−1} ··· c_0)_B;
```

Note: At every step, the value of *carry* lies between 0 and $B - 1$, and the value of *tmp* lies between 0 and $B^2 - 1$.
Time Complexity: $O(\|a\| \cdot \|b\|)$, since the algorithm involves $k \cdot \ell$ multiplications.
Space Complexity: $O(\|a\| + \|b\|)$, since we store the digits of a, b, and c.

Function 2.6: Decimal to Binary Conversion - *DecToBin()*

This function converts a decimal number n into its binary equivalent by repeatedly dividing the decimal number by 2 and recording the remainders.

Input: n (a decimal number)
Output: b (binary representation of n)

```
1 Function DecToBin(n):
2     b ← empty string;
3     while n > 0 do
4         r ← n mod 2;
5         n ← ⌊n/2⌋;
6         b ← r + b;
7     return b;
```

Time Complexity: $O(\log n)$, as the number of iterations is proportional to the number of bits in n.
Space Complexity: $O(n)$, storing our input n.

Example: Converting 89 to binary given the above function:

$$\begin{aligned}
89_{10} \div 2 &= 44 \quad \text{rem } 1, \longleftarrow \textbf{ LSB} \\
44_{10} \div 2 &= 22 \quad \text{rem } 0, \\
22_{10} \div 2 &= 11 \quad \text{rem } 0, \\
11_{10} \div 2 &= 5 \quad \text{rem } 1, \\
5_{10} \div 2 &= 2 \quad \text{rem } 1, \\
2_{10} \div 2 &= 1 \quad \text{rem } 0, \\
1_{10} \div 2 &= 0 \quad \text{rem } 1. \longleftarrow \textbf{ MSB}
\end{aligned}$$

Thus, $89_{10} = 1011001_2$.

Theorem 2.2: Time Complexity of Basic Arithmetic Operations

We generalize the time complexity to a and b as n-bit integers.

(i) **Addition & Subtraction:** $a \pm b$ in time $O(n)$.

(ii) **Multiplication:** $a \cdot b$ in time $O(n^2)$.

(iii) **Quotient Remainder** quotient $q := \left\lfloor \frac{a}{b} \right\rfloor : b \neq 0, a > b$; and remainder $r := a \mod b$ has time $O(n^2)$.

2.3 Computational Efficiency

Theorem 3.1: Binary Length of a Number - $\|a\|$

The binary length of an integer a_{10} in binary representation, is given by:

$$\|a\| := \begin{cases} \lfloor \log_2 |a| \rfloor + 1 & \text{if } a \neq 0, \\ 1 & \text{if } a = 0, \end{cases}$$

as $\lfloor \log_2 |a| \rfloor + 1$ correlates to the highest power of 2 required to represent a.

Example: Think about base 10 first. Let there be a 9 digit number $d = 684,301,739$. To reach 9 digits takes 10^8; The exponent plus 1 yields $\|d\|$. Hence, $\lfloor \log_{10} d \rfloor + 1$ is $\|d\|$.

Now, let there be a 7 digit binary number $b = 1001000$, which expanded is:

$$(1 \cdot 2^6) + (0 \cdot 2^5) + (0 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 72,$$

Taking 6 powers of 2 to reach 72, we add 1 to get $\|b\| = 7$. Hence, $\|b\| = \lfloor \log_2 b \rfloor + 1$.
Additionally, if $a = 0_2$ then $\|a\| = 1$. as $a^0 = 1$.

Theorem 3.2: Splitting Higher and Lower Bits

Let a be a binary number with n bits. We can split a into two numbers A_1 and A_0 with $n/2$ bits each, representing the first and second halves respectively. Where:

$$A_1 := \frac{a}{2^{\lceil n/2 \rceil}} \quad \text{and} \quad A_0 := a \bmod 2^{\lceil n/2 \rceil}$$

Example: Let's start with base 10. To achieve $A_1 = 7455$ and $A_0 = 62,010$, for $a = 745,562,010$. we take the length $\|a\| := \lfloor \log_{10}(745,562,010) \rfloor + 1 = 9$, as $10^8 \leq 745,562,010 < 10^9$. Then:

$$A_1 = \frac{745,562,010}{10^{\lceil 9/2 \rceil}} = 7455, \quad \text{and} \quad A_0 = 745,562,010 \bmod 10^{\lceil 9/2 \rceil} = 62,010$$

as $10^5 \leq 62,010 < 10^6$. Likewise to finding the remainder in base 2, we can use the same bit shifting technique for base 10 (2.1). We see,

$$[745,562,010]_{10} \text{ right shift by } 5, [000,007,455]_{10} \; 62,010.$$

Hence, $62,010$ is pushed out, and our remainder. Then, we can apply the same technique to base 2. Let $a = 1111\ 1111\ 1001\ 1001_2$. We have $\|a\| := 16$, then:

$$A_1 = \frac{1111\ 1111\ 1001\ 1001_2}{2^{\lceil 16/2 \rceil}} = 1111\ 1111_2, \text{ and } A_0 = 1111\ 1111\ 1001\ 1001_2 \bmod 2^{\lceil 16/2 \rceil} = 1001\ 1001_2$$

Scenario - *Divide and Conquer Multiplication*: We are to compute,

$$A_1 2^{\lceil n/2 \rceil} + A_0 =: a \quad \times \quad b := B_1 2^{\lceil n/2 \rceil} + B_0.$$

Then we have,

$$\begin{aligned} a \cdot b &= (A_1 2^{\lceil n/2 \rceil} + A_0)(B_1 2^{\lceil n/2 \rceil} + B_0) \\ &= (A_1 2^{\lceil n/2 \rceil})(B_1 2^{\lceil n/2 \rceil}) + (A_1 2^{\lceil n/2 \rceil})B_0 + (B_1 2^{\lceil n/2 \rceil})A_0 + A_0 B_0 \\ &= (A_1 B_1)2^n + (A_1 B_0 + B_1 A_0)2^{\lceil n/2 \rceil} + A_0 B_0. \end{aligned}$$

We need to compute 4 products, (A_1B_1), $(A_1B_0), (B_1A_0)$, and (A_0B_0). We now attempt to solve them independently:

Function 3.1: Multiplication of n-bit Integers - *Multiply()*

Let a and b be n-bit integers of base 2. This algorithm recursively computes the product of a and b using a straightforward divide-and-conquer approach, without using Karatsuba's optimization.

Input: n, a, b (where a and b are n-bit integers)
Output: The product $a \times b$

```
1  Function Multiply(n, a, b):
2      if n < 2 then
3          return the result of grade-school multiplication for a × b;
4      else
5          A_1 ← a ÷ 2^{n/2};  A_0 ← a mod 2^{n/2};
6          B_1 ← b ÷ 2^{n/2};  B_0 ← b mod 2^{n/2};
7          p_1 ← Multiply(n/2, A_1, B_1);
8          p_2 ← Multiply(n/2, A_1, B_0);
9          p_3 ← Multiply(n/2, A_0, B_1);
10         p_4 ← Multiply(n/2, A_0, B_0);
11         return p_1 · 2^n + (p_2 + p_3) · 2^{n/2} + p_4;
```

Time Complexity: $O(n^2)$, as in our master method $T(n) = 4T(n/2) + O(n)$, Theorem (1.5).
Space Complexity: $O(n)$, storing $n + n$ bits for a and b, while we track $O(\log_2 n)$ depth in the recursion stack.

We appear to make no improvement, however there's a small trick to reduce the number of multiplications. We continue on the next page.

Observe our full term, $c := (A_1B_1)2^n + (A_1B_0 + B_1A_0)2^{\lceil n/2 \rceil} + A_0B_0$. Say we computed another term,

$$z := (A_1 + A_0)(B_1 + B_0) = (A_1B_1) + (A_1B_0) + (B_1A_0) + (A_0B_0).$$

Notice how z also contains (A_1B_1) and (A_0B_0), which are also in c. Say $m = (A_1B_0) + (B_1A_0)$. Let $x := (A_1B_1)$ and $y := (A_0B_0)$ then $z - x - y = m$. This reduces the number of multiplications to 3, as we only compute (A_1B_1), (A_0B_0) once, and then z.

We employ the above strategy, which is **Karatsuba's multiplication algorithm**:

Function 3.2: Karatsuba's Multiplication Algorithm - *KMul()*

Let a and b be n-bit integers of base 2. This algorithm recursively computes the product of a and b using a divide-and-conquer approach.

Input: n, a, b (where a and b are n-bit integers)
Output: The product $a \times b$

```
1  Function Multiply(n, a, b):
2      if n < 2 then
3          return the result of grade-school multiplication for a × b;
4      else
5          A_1 ← a ÷ 2^{n/2};  A_0 ← a mod 2^{n/2};
6          B_1 ← b ÷ 2^{n/2};  B_0 ← b mod 2^{n/2};
7          x ← Multiply(n/2, A_1, B_1);
8          y ← Multiply(n/2, A_0, B_0);
9          z ← Multiply(n/2, A_1 + A_0, B_1 + B_0);
10         return x · 2^n + (z − x − y) · 2^{n/2} + y;
```

Time Complexity: $O(n^{\log_2 3}) \approx O(n^{1.585})$, as in our master method $T(n) = 3T(n/2) + O(n)$, Theorem (1.5).
Space Complexity: $O(n)$.

— 3 —

Evaluating Recurrences

3.1 Inductive Analysis

Recursion employs techniques of repeatedly shrinking the problem to solve its smaller halves. Often found in sorting problems, processing trees, and geometric problems.

Theorem 1.1: MergeSort

Given an array A of size n, we sort via the following:

(i.) Recursively: Split the array into two halves and wait for a return.

(ii.) Base-case: If the array has one element, return it.

(iii.) Merge: receive two halves and merge them.

The final result is a sorted array.

Theorem 1.2: QuickSort

Given an array A of size n, we sort via the following:

(i.) Choose a pivot, set i to the start and j at the end of the array.

(ii.) Increase i until $A[i] > pivot$ and decrease j until $A[j] < pivot$.

- If $A[i] < A[j]$, swap $A[i]$ and $A[j]$.
- Repeat until $i > j$, return j as the new pivot.

(iii.) The new pivot creates two sub-arrays, repeat the process on each sub-array.

The final result is A sorted in-place (on the original array without a temporary helper array).

Tip: Demos for **MergeSort**: https://www.youtube.com/watch?v=4VqmGXwpLqc and **QuickSort**: https://www.youtube.com/watch?v=Hoixgm4-P4M.

Function 1.1: Mergesort - `MSORT(A)`

Input: Array A and temporary array $temp$ of n elements.
Output: Nothing is returned, the array A is sorted by reference.

MSORT function*(A,temp,i,j)*:

```
1  if i ≥ j then
2  |  return;
3  mid ← (i + j)/2;
4  MSORT(A, temp, i, mid); // Left subarray
5  MSORT(A, temp, mid + 1, j); // Right subarray
6  merge(A, temp, i, mid, mid + 1, j); // Merge both halves
```

Merge function*(A,temp,i,leftEnd,j,rightEnd)*:

```
1  i ← lefti; // Left subarray
2  j ← righti; // Right subarray
3  k ← lefti; // Temporary array
4  while i ≤ leftEnd and j ≤ rightEnd do
5  |  if A[i] < A[j] then
6  |  |  temp[k] ← A[i]; i ← i + 1;
7  |  else
8  |  |  temp[k] ← A[j]; j ← j + 1;
9  |  k ← k + 1;
10 while i ≤ leftEnd do
11 |  temp[k] ← A[i]; i ← i + 1;
12 |  k ← k + 1;
13 while j ≤ rightEnd do
14 |  temp[k] ← A[j]; j ← j + 1;
15 |  k ← k + 1;
16 for i = lefti to rightEnd do
17 |  A[i] ← temp[i]; // Copy back sorted elements
```

Time Complexity: $O(n \log n)$ in all cases.
Space Complexity: $O(n \log n)$ for the temporary array and $O(\log n)$ for the recursion stack.

Function 1.2: Quicksort - `QSORT(A,high,low)`

Input: Array A of n elements.
Output: Sorted array A in ascending order.
Initial Call: $QSORT(A, 0, n-1)$

QSORT function*(A,low,high)*

```
1  if low < high then
2  |  pivot ← partition(A, low, high);
3  |  QSORT(A, low, pivot); // Left of the pivot
4  |  QSORT(A, pivot + 1, high); // Right of the pivot
```

Partition function*(A,low,high)*:

```
1  pivot ← A[⌊(low + high)/2⌋];
2  i ← low − 1;
3  j ← high + 1;
4  while i < j do
5  |  repeat
6  |  |  if A[i] ≥ pivot then
7  |  |  |  break;
8  |  until i ← i + 1;
9  |  repeat
10 |  |  if A[j] ≤ pivot then
11 |  |  |  break;
12 |  until j ← j − 1;
13 |  if i < j then
14 |  |  A.swap(i, j);
15 return j; // Return the index of the partition
```

Time Complexity: Average case $O(n \log n)$, Worst case $O(n^2)$.
Space Complexity: $O(n)$ for input. The sorting is done in place, and the recursion stack takes $O(\log n)$ space in the average case. Worst case space complexity is $O(n)$.

Theorem 1.3: Worst Cases - Merge and Quick Sort

- **Merge Sort:** independent of the input, always $O(n \log n)$.
- **Quick Sort:** If the array is sorted in ascending/descending order, the pivot is the smallest/largest element, respectively. This results in $O(n^2)$ time complexity, as each partition is of size $n-1$.

Let us examine merge sort at a high-level.

Function 1.3: Mergesort - `MSORT(A)`

Input: Array A and temporary array $temp$ of n elements.
Output: Nothing is returned, the array A is sorted by reference.

MSORT function:

```
1 if i ≥ j then
2 |   return;
3 mid ← (i + j)/2;
4 MSORT(A, temp, i, mid); // Left subarray
5 MSORT(A, temp, mid + 1, j); // Right subarray
6 merge(A, temp, i, mid, mid + 1, j); // Merge both halves
```

Proof 1.1: Merge Sort - Time Complexity

We make the following observations, generalizing n to each recursive call:

(i.) We make 2 recursive calls.

(ii.) Each recursive call cuts the array in half, $n/2$.

(iii.) We do $\Theta(n)$ work to merge the two halves, returning up the stack.

We define a general form for our traversals as function $T(n)$:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

number of rucursive calls — sub-divisions each frame — work each stack frame

For merge sort, we have $T(n) = 2 \cdot T(n/2) + \Theta(n)$. This means we i with input $T(n)$ and then our first recursive call is $T(n/2)$, the calls from there are $T(n/4)$, $T(n/8)$, and so on. Therefore, at any given layer k, we have 2^k calls, with each input $T(n/2^k)$. We stop when $n/2^k = 1$, which is $k = \log n$. Since $2^{\log_2 n} = n$, then $\left(\frac{n}{2^{\log_2 n}}\right) = 1$. Thus, the depth of our recursion is $\log n$, and n work is done when unraveling the stack, hence $O(n \log n)$.

■

To illustrate the above proof, we can draw a recursion tree for merge sort:

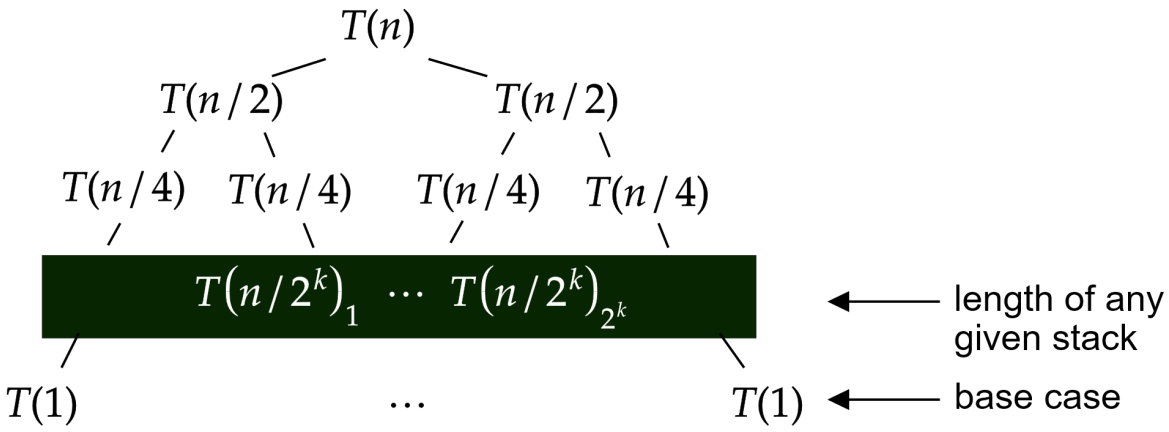


Figure 3.1: Recursion Tree for Merge Sort

Function 1.4: Quicksort - `QSORT(A)`

Input: Array A of n elements.
Output: Sorted array A in ascending order.

QSORT function:

```
1 if low < high then
2 |   pivotIndex ← partition(A, low, high);
3 |   QSORT(A, low, pivotIndex); // Left of the pivot
4 |   QSORT(A, pivotIndex + 1, high); // Right of the pivot
```

Proof 1.2: Quick Sort - Time Complexity

First does $\Theta(n)$ work to partition, then makes 2 recursive calls, and each recursive call has on average $n/2$ elements. We define a general form for our traversals as function $2T(n/2)+\Theta(n)$. Thus $\log n$ levels of recursion, each taking $\Theta(n)$ time to partition, hence $O(n \log n)$.

In worst case, $2T(n-1)+\Theta(n)$. Then we have n levels of recursion, each taking $\Theta(n)$ time to partition, hence $O(n^2)$.

■

Theorem 1.4: Proving Correctness of Recursive Functions

To prove correctness of a recursive function, we need to show:

1. **Base Case:** The base case is correct.
2. **Inductive Hypothesis:** Assume k input sizes are correct, where $k < n$, we assume the recursive calls return the correct result.
3. **Inductive Step:** Show that the problem reduces to the base case, and that intermediate steps other than the recursive calls are correct.

If all three conditions are met, then the function is correct for all n.

Theorem 1.5: Master Method for Recursive Time Complexity

The master method is a general technique for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n^d)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is a given function. We consider degree d of $f(n)$:

$$\begin{aligned} d > \log_b a &\implies T(n) = \Theta(n^d) \\ d < \log_b a &\implies T(n) = \Theta(n^{\log_b a}) \\ d = \log_b a &\implies T(n) = \Theta(n^{\log_b a} \log n) \end{aligned}$$

Tip: Extended Version of the Master Method:

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T(b_i n + h_i(n))$$

where $h_i(n) = O\left(\frac{n}{\log^2 n}\right)$. This is the **Akra-Bazzi Method**:
Link: https://en.wikipedia.org/wiki/Akra%E2%80%93Bazzi_method.

Examples:

- $T(n) = 2T\left(\frac{n}{2}\right) + n^3$: $(a = 2;\ b = 2;\ d = 3;)$ then $(\log_2 2 = 1 < 3)$ hence $T(n) = \Theta(n^3)$.
- $T(n) = 5T\left(\frac{n}{2}\right) + n^2$: $(a = 5;\ b = 2;\ d = 2;)$ then $(\log_2 5 \approx 2.32 > 2)$ hence $T(n) = \Theta(n^{\log_2 5})$
- $T(n) = 16T\left(\frac{n}{4}\right) + n^2$: We have $d := 2$ and $(log_4 16 = 2 = d)$ hence $T(n) = \Theta(n^{log_4 16} \log n)$

Function 1.5: Mergesort - `MSORT(A)`

Input: Array A and temporary array $temp$ of n elements.
Output: Nothing is returned, the array A is sorted by reference.

MSORT function:

```
1 if i ≥ j then
2 |   return;
3 mid ← (i + j)/2;
4 MSORT(A, temp, i, mid); // Left subarray
5 MSORT(A, temp, mid + 1, j); // Right subarray
6 merge(A, temp, i, mid, mid + 1, j); // Merge both halves
```

Proof 1.3: Correctness of Merge Sort

By strong induction on the input size n:

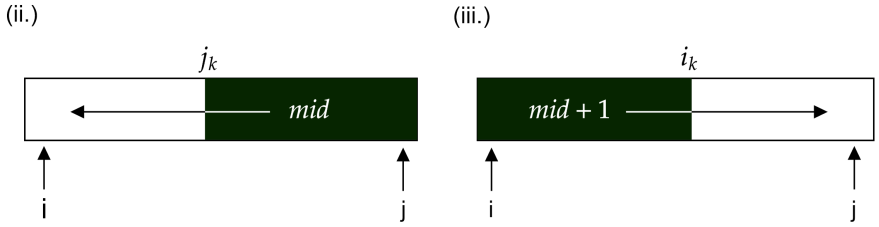
1. **Base Case:** If $i \geq j$, then the array is of size 1, and is already sorted.
2. **Inductive Hypothesis:** Assume that Merge Sort correctly sorts arrays of sizes k where $1 \leq k < n$.
3. **Inductive Step:** Our two recursive calls follow,

   (i.) $mid \leftarrow \lfloor (i + j)/2 \rfloor$

   (ii.) $MSORT(A, temp, i, mid)$

   (iii.) $MSORT(A, temp, mid + 1, j)$

Suppose (ii.) and (iii.) do not reach the base case. Then in (ii.) $mid >= j$ and in (iii.) $mid + 1 <= i$, which contradicts, as $mid := \lfloor (i + j)/2 \rfloor$, then $i \leq mid \leq j$.



Then in (ii.), the right bound mid keeps decreasing, and in (iii.), the left bound mid keeps increasing. This shirks both sub-arrays until both bounds meet. The merge function sorts after both calls, taking the next biggest in the sub-arrays, placing it in the temporary array, and copying back to the original array. Hence, by induction, the function is correct. ■

Theorem 1.6: Iterative Substitution Method (plug-and-chug)

Given a function $T(n)$ which has a recurrence relation—meaning it calls upon itself in its own definition—we may repeatedly substitute such self-references back into $T(n)$.

Given that $T(n)$ properly subdivides to a base case $T(x)$, we may derive some pattern which illustrates the state of $T(n)$ at some depth k within the recurrence. Doing so, we identify what makes our k_{th} expression hit $T(x)$.

Proof 1.4: Iterative Substitution Method (plug-and-chug) - MergeSort

Merge sort has the recurrence relation $T(n) = 2T(n/2) + \Theta(n)$, for which the base case is $T(1)$. We substitute $T(n/2)$ into $T(n)$:

$$\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n); \quad \text{and} \quad T(n/2) = 2T(n/2^2) + \Theta(n/2); \\
T(n) &= 2[\quad\quad] + \Theta(n); \text{ Prepare to substitute the recurrence } T(n/2) \\
&= 2\left[2T(n/2^2) + \Theta(n/2)\right] + \Theta(n); \text{ we evaluate } 2 \cdot \Theta(n/2) \text{ as } \Theta(2n/2) \\
&= 2^2 T(n/2^2) + \Theta(n) + \Theta(n); \text{ Simplified}
\end{aligned}$$

We won't fully simplify to observe how the recurrence builds. We continue, evaluating $T(n/2^2)$:

$$\begin{aligned}
T(n/2^2) &= 2T(n/2^3) + \Theta(n); \\
T(n) &= 2^2\left[2T(n/2^3) + \Theta(n/2^2)\right] + \Theta(n) + \Theta(n); \text{substitute again} \\
&= 2^3 T(n/2^3) + \Theta(n) + \Theta(n) + \Theta(n); \text{ Simplified}
\end{aligned}$$

We identify the pattern for the k_{th} substitution:

$$T(n) = 2^k T(n/2^k) + k \cdot \Theta(n); \text{ General form}$$

Now we identify what makes our recurrence $T(n/2^k) = T(1)$, i.e., where is $n/2^k = 1$, then $n = 2^k$, and $\log_2 n = k$. We plug this back into our general form:

$$\begin{aligned}
T(n) &= 2^{\log_2 n} T(n/2^{\log_2 n}) + \log_2 n \cdot \Theta(n); \text{ Substituting } k \\
&= n\Theta(1) + \log_2 n \cdot \Theta(n); \text{ Where } T(1) = \Theta(1) \\
&= \Theta(n) + \Theta(n)\log_2 n; \\
&= \Theta(n \log n);
\end{aligned}$$

Hence, merge sort has a time complexity of $O(n \log n)$.

■

Tip: Live Demo: https://youtu.be/Ob8SM0fz6p0?si=Z4PZQDWOXa7deEHA