problem statement.

Typically, we assume that basic operations on natural numbers (e.g., adding or multiplying two natural numbers together) are performed in constant time. In practice, this assumption is correct whenever we restrict ourselves to natural numbers with some maximum size (e.g., 64 bit natural numbers, for which basic operations are supported directly by modern processors). Applications such as cryptography often work with huge natural numbers, however (e.g., 4048 bit values, which can hold a maximum of $\approx 3.7 \cdot 10^{1218}$). Hence, for these applications we can no longer assume that operations on natural numbers are in constant time: these applications require the development of efficient algorithms even for basic operations on natural numbers.

Consider two n-digit natural numbers $A = a_1 \dots a_n$ and $B = b_1 \dots b_n$ written in base 10: the digits $a_1 \dots a_n$ and $b_1 \dots b_n$ each have a value in $0 \dots 9$. For example, if $n = 4$, then we could have $A = 3456$, $B = 9870$, in which case $a_1 = 3, a_2 = 4, a_3 = 5, a_6 = 6, b_1 = 9, b_2 = 8, b_3 = 7, b_4 = 0$.

ⓘ 1.1

Write an algorithm `ADD(A, B)` that computes $A + B$ in $\Theta(n)$. Explain why your algorithm is correct and the runtime complexity is $\Theta(n)$.

Assumption: one converts A and B into two arrays of n integers, $A = [a_1 \dots a_n]$ and $B = [b_1 \dots b_n]$.

Algorithm ADD(A, B)

Input: $A := [a_1 \dots a_n]$
Input: $B := [b_1 \dots b_n]$
  $C \leftarrow [\,]$ where $|C| = n + 1$
  $carry \leftarrow 0$
  $i \leftarrow n - 1$
  while $i \geq 0$ **do**
    $C[i+1] \leftarrow (a_i + b_i + carry) \mod 10$
    $carry \leftarrow \lfloor (a_i + b_i + carry)/10 \rfloor$
    $i \leftarrow i - 1$
  end while
  $C[0] \leftarrow carry$
  if $C[0] == 0$ **then**
    $C \leftarrow C[1 \dots n]$
  end if
Output: C

Runtime complexity: $\Theta(n)$

- L1 takes $\Theta(n)$ time to initialise.
- `while` loop iterates n times, each iteration perform constant time operations (additions, modulo, division) in $\Theta(1)$ time.
- Finally, the adjustment of the output array C takes $\Theta(1)$ time.

Thus, total runtime complexity is $\Theta(n)$.

Correctness:

Invariants:

$$
\begin{aligned}
0 \leq i \leq n-1, i+2 \leq j \leq n \wedge c_{n-1} = 0 \\
c = \lfloor \frac{\sum_{k=i+1}^{n-1}(a_k + b_k + c_k)}{10^{n-k-1}} \rfloor \mod 10 \\
C[i+1] = (a_i + b_i + c) \mod 10 \\
C[j] = ((a_{j-1} + b_{j-1} + c_{j-1}) \mod 10)
\end{aligned}
$$

where c defines as the carry value resulting from the addition.

bound function $f(i) = |A| - i$ starts at $|A|, |A| \geq 0$

Proof

Base case: $i = n-1$ (*L2,3*)

Invariant for carry holds, as $c_i = c_{n-1} = 0$

Now we will prove these invariants still hold til reach the end of `m-th` loop:

Assuming the invariants hold at the start of `m-th` loop, or:

$$
\begin{aligned}
0 \leq m \leq n-1 \\
c_m = \lfloor \frac{\sum_{k=m}^{n-1}(a_k + b_k + c_k)}{10^{n-k-1}} \rfloor \mod 10 \\
C[m+1] = (a_m + b_m + c_m) \mod 10 \\
C[j] = ((a_{j-1} + b_{j-1} + c_{j-1}) \mod 10)
\end{aligned}
$$

L4-7: The `while` loop.

- Carry forward invariants holds
  $c_{m-1} = c_{\text{new}} = \lfloor \frac{(a_m + b_m + c_m)}{10} \rfloor \mod 10$

- $C[m+1] = (a_m + b_m + c_m) \mod 10$, or $C[m+1]$ holds correct digits after addition of a_m, b_m and carry c_m
- $f(i)$ strictly decreases after each iteration, $i_{\text{new}} := i + 1$

Therefore the invariants holds.

> ⓘ **1.2**
>
> What is the runtime complexity of this algorithm in terms of the number of digits in A and B?

Runtime complexity is $\Theta(n^2)$, where n is the number of digits in A and B.

For each digits of B, it multiply every digits of A, which results in n^2 operations.

Each addition operation takes at most $2n$ digit additions, and we perform n of these additions, therefore resulting in $O(n^2)$ time.

Overall, pen-and-paper addition of two n-digit numbers takes $\Theta(n^2)$ time.

> ⓘ **1.3**
>
> Let C be an n-digit number with $n = 2m$. Hence, $C = C_{\text{high}} \cdot 10^m + C_{\text{low}}$ where C_{high} the first m digits of C and C_{low} is the remaining m digits of C. For example, if $n = 4$, $A = 3456$, $B = 9870$, then $m = 2$ and

$$
\begin{aligned}
A &= A_{\text{high}} \cdot 10^m + A_{\text{low}}, & A_{\text{high}} &= 34, & A_{\text{low}} &= 56 \\
B &= B_{\text{high}} \cdot 10^m + B_{\text{low}}, & B_{\text{high}} &= 98, & B_{\text{low}} &= 70
\end{aligned}
$$

Using the breakdown of a number into their high and low part, one notices the following

$$
\begin{aligned}
A \times B &= (A_{\text{high}} \cdot 10^m + A_{\text{low}}) \cdot (B_{\text{high}} \cdot 10^m + B_{\text{low}}) \\
&= A_{\text{high}} \times B_{\text{high}} \cdot 10^{2m} + (A_{\text{high}} \times B_{\text{low}} + A_{\text{low}} \times B
\end{aligned}
$$

Here is the following recursive algorithm `BREAKSDOWNMULTIPLY(A, B)` that computes $A \times B$:

Algorithm BREAKSDOWNMULTIPLY(A, B)

Input: A and B have $n = 2m$ digits

```
if n = 1 then
    return a_1 × b_1
else
    hh := BREAKSDOWNMULTIPLY(A_high, B_high)
    hl := BREAKSDOWNMULTIPLY(A_high, B_low)
    lh := BREAKSDOWNMULTIPLY(A_low, B_high)
    ll := BREAKSDOWNMULTIPLY(A_low, B_low)
    return hh · 10^{2m} + (hl + lh) · 10^m + ll
end if
return A × B
```

Prove that algorithm `BREAKSDOWNMULTIPLY(A, B)` is correct.

The proposed `BREAKSDOWNMULTIPLY(A, B)` is a variant of Karatsuba's algorithm.

Base case: $m = 1 \implies n = 2$, which implies $A \times B$ are correct (multiplication of two two-digits number).

Through recursions, at any level k, $k = \log_2 n$, $n_k = 2^k \cdot m$, one would observe:

- $A_k = A_{\text{high}_k} \cdot 10^{m_k} + A_{\text{low}_k}$
- $B_k = B_{\text{high}_k} \cdot 10^{m_k} + B_{\text{low}_k}$

The recursive call hh_k, hl_k, lh_k, ll_k correctly computes the product of $A_k \times B_k$ til the base case.

The combination of the products is proven through previous math steps, therefore, the algorithm is correct.

> ⓘ **1.4**
>
> Give a recurrence $T(n)$ for the runtime complexity of `BREAKSDOWNMULTIPLY(A, B)` Explain each term in the recurrence.
>
> Draw a recurrence tree for $T(n)$ and use this recurrence tree to solve the recurrence $T(n)$ by proving that $T(n) = \Theta(f(n))$ for some function $f(n)$
>
> What is the runtime complexity of `BREAKSDOWNMULTIPLY(A, B)`? Do you expect this algorithm to be faster than the pen-and-paper multiplication algorithm?
> *Hint: Feel free to assume that $n = 2^k, k \in \mathbb{N}$. Feel free to assume*

that we can add two v-digit number in $\Theta(v)$ (e.g., using `ADD`) and that we can multiply a v-digit number with 10^w in $\Theta(v+w)$.

For two n digits number A and B, the recurrent $T(n)$ is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 4T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- The base case when $n = 1$ is $\Theta(1)$, as it only performs a single digit multiplication, without no recursive calls.
- The recursive case when $n > 1$ performs 4 recursive calls, each with $n/2$ digits, each on number half the size of original input (since $n = 2m$), hence $4T(n/2)$.
- $\Theta(n)$ is the linear time complexity adding the products of the recursive calls, per our assumption that we can multiply a v-digit number with 10^w in $\Theta(v+w)$.

The recurrence tree for $T(n)$ is:

```
T(n)
├── T(n/2)
|    ├── T(n/4)
|    |    ├── T(n/8)
|    |    |    ├── ...
|    |    |    ...
|    |    |    ├── ...
|    |    |    └── ...
|    |   ...
|    |    └── T(n/8)
```

```
│         ├── T(n/4)
│         │         ├── ...
│         │         ├── ...
│         │         ├── ...
│         │         └── ...
│         ├── T(n/4)
│         │         ├── ...
│         │         ├── ...
│         │         ├── ...
│         │         └── ...
│         └── T(n/4)
│                   ├── ...
│                   ├── ...
│                   ├── ...
│                   └── ...
├── T(n/2)
│         ├── ...
│         ├── ...
│         ├── ...
│         └── ...
├── T(n/2)
│         ├── ...
│         ├── ...
│         ├── ...
│         └── ...
└── T(n/2)
          ├── ...
          ├── ...
          ├── ...
          └── ...
```

- The total number of nodes at depth k is 4^k, since each level of recursion calls the function four times.
- Work done at level k is $4^k \cdot n/2^k = 2^k \cdot n$, since work done per depth is n times the number of nodes add that depth.
- Depth of the tree is $\log_2 n$, since the input size is halved at each level.

Therefore, one can solve for $T(n)$:

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log_2(n)} 2^k \cdot n \\ &= n \cdot \sum_{k=0}^{\log_2(n)} 2^k \\ &= n \cdot \frac{2^{\log_2(n)+1} - 1}{2-1} \\ &= n \cdot (2n-1) \\ &= 2n^2 - n \\ &= \Theta(n^2) \end{aligned}$$

Thus the runtime complexity of `BREAKSDOWNMULTIPLY(A, B)` is quadratic, $\Theta(n^2)$.

From here, the algorithm is the same as the pen-and-paper multiplication algorithm, which also takes $\Theta(n^2)$ time.

1.5

One can observe

$(A_{\text{high}} + A_{\text{low}}) \times (B_{\text{high}} + B_{\text{low}}) = A_{\text{high}} \times B_{\text{high}} + A_{\text{high}} \times$

Hence by rearranging terms, one can conclude that

$$A_{\text{high}} \times B_{\text{low}} + A_{\text{low}} \times B_{\text{high}} = (A_{\text{high}} + A_{\text{low}}) \times (B_{\text{high}} +$$

Based on conclusion above, $A \times B$ can be seen as:

$$\begin{aligned}
A \times B &= (A_{\text{high}} \cdot 10^m + A_{\text{low}}) \times (B_{\text{high}} \cdot 10^m + B_{\text{low}}) \\
&= A_{\text{high}} \times B_{\text{high}} \cdot 10^{2m} + A_{\text{high}} \times B_{\text{low}} \cdot 10^m + A_{\text{low}} \\
&= A_{\text{high}} \times B_{\text{high}} \cdot 10^{2m} + (A_{\text{high}} \times B_{\text{low}} + A_{\text{low}} \times B \\
&= A_{\text{high}} \times B_{\text{high}} \cdot 10^{2m} + (((A_{\text{high}} + A_{\text{low}}) \times (B_{\text{high}}
\end{aligned}$$

The final rewritten form of $A \times B$ only requires three multiplication terms, namely
$A_{\text{high}} \times B_{\text{high}}, A_{\text{low}} \times B_{\text{low}}, (A_{\text{high}} + A_{\text{low}}) \times (B_{\text{high}} + B_{\text{low}}$

Use the observation to construct a recursive multiplication `SMARTMATHSMULTIPLY(A, B)` that only perform three recursive multiplications. Argue why `SMARTMATHSMULTIPLY(A, B)` is correct.

Algorithm SMARTMATHSMULTIPLY(A, B)

Input: A and B have $n = 2m$ digits

  if $n = 1$ **then**
    return $a_1 \times b_1$
  else
    $hh :=$ SMARTMATHSMULTIPLY$(A_{\text{high}}, B_{\text{high}})$
    $ll :=$ SMARTMATHSMULTIPLY$(A_{\text{low}}, B_{\text{low}})$
    $mid :=$ SMARTMATHSMULTIPLY$(A_{\text{high}} + A_{\text{low}}, B_{\text{high}} + B_{\text{low}})$
    return $hh \cdot 10^{2m} + (mid - hh - ll) \cdot 10^m + ll$
  end if

return $A \times B$

The proposed `SMARTMATHSMULTIPLY(A, B)` is *the basis* of Karatsuba's algorithm.

Base case: $n = 1$, which implies $A \times B$ are correct (multiplication of two single digit number).

Assume that `SMARTMATHSMULTIPLY(A, B)` correctly computes the product of $A \times B$ for A, B with lest than n digits.

The following invariants hold per recursive call:

- $A = A_{\text{high}} \cdot 10^m + A_{\text{low}} \wedge B = B_{\text{high} \cdot 10^m + B_{\text{low}}}$ where $m = \frac{n}{2}$ (true from problem statement and $n = 2^k$)
- recursive call computes P_1, P_2, P_3 correctly, where $P_1 = A_{\text{high}} \times B_{\text{high}}, P_2 = A_{\text{low}} \times B_{\text{low}}, P_3 = (A_{\text{high}} + A_{\text{low}}) \times$ for numbers fewer than n digits (from induction hypothesis)
- combination invariants: $P_4 = P_3 - P_2 - P_1 \wedge A \times B = P_1 \cdot 10^{2m} + P_4 \cdot 10^m + P_2$ (true from previous statement)

Thus, the algorithm is correct.

1.6

Give a recurrence $T(n)$ for the runtime complexity of `SMARTMATHSMULTIPLY(A, B)` Explain each term in the recurrence.

Solve the recurrence $T(n)$ by proving that $T(n) = \Theta(f(n))$ for some function $f(n)$. Use any methods that you find comfortable with.

What is the runtime complexity of `SMARTMATHSMULTIPLY(A, B)`? Do you expect this algorithm to be faster than the pen-and-paper multiplication algorithm?
Hint: Feel free to assume that $n = 2^k, k \in \mathbb{N}$. Feel free to assume that we can add two v-digit number in $\Theta(v)$ (e.g., using `ADD`) and that we can multiply a v-digit number with 10^w in $\Theta(v + w)$.

For two n digits number A and B, the recurrent $T(n)$ is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 3T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- The base case when $n = 1$ is $\Theta(1)$, as it only performs a single digit multiplication, without no recursive calls.
- The recursive case when $n > 1$ performs 3 recursive calls, each with $n/2$ digits, each on number half the size of original input (since $n = 2m$), hence $3T(n/2)$.
- $\Theta(n)$ is the linear time complexity adding the products of the recursive calls, per our assumption that we can multiply a v-digit number with 10^w in $\Theta(v + w)$.

Using `Master Theorem`, we can solve for $T(n)$, with $a = 3, b = 2, f(n) = \Theta(n) = n^{\log_2 3}$.

> The master theorem states that if $f(N) = \Theta(N^{\log_b a} \log^k(N))$, with $k > 0$, then $T(N) = \Theta(N^{\log_b a} \log^{k+1} N)$.

Thus $T(n) = \Theta(n^{\log_2 3} \log(n)) = \Theta(n^{\log_2 3})$

> ♨ **Runtime complexity of `SMARTMATHSMULTIPLY(A, B)`**
>
> $\Theta(n^{\log_2 3}) \approx \Theta(n^1.585)$
>
> This algorithm is expected to be faster than the pen-and-paper multiplication algorithm, which also takes $\Theta(n^2)$ time.