Representing Big Integers

**Multiple-precision** integers require more storage than a single machine word like an ‘int’.

Remember: why are these important computationally?

Example: 4391354067575026 represented as an array:
- In base $B = 10$: 
  \[ [6, 2, 0, 5, 7, 5, 7, 6, 0, 4, 5, 3, 1, 9, 3, 4] \]
- In base $B = 256$: 
  \[ [242, 224, 71, 203, 233, 153, 15] \]

Base of representation

General form of a multiple-precision integer:
\[ d_0 + d_1 B + d_2 B^2 + d_3 B^3 + \cdots + d_{n-1} B^{n-1}. \]

Does the choice of base $B$ matter?

Addition

How would you add two $n$-digit integers?
- Remember, every digit is in a separate machine word.
- How big can the “carries” get?
- What if the inputs don’t have the same size?
- How fast is your method?
Standard Addition

```python
def add(X, Y, B):
    carry = 0
    A = zero-filled array of length (len(X) + 1)
    for i in range(0, len(Y)):
        carry, A[i] = divmod(X[i] + Y[i] + carry, B)
    for i in range(len(Y), len(X)):
        carry, A[i] = divmod(X[i] + carry, B)
    A[len(X)] = carry
    return A
```

Linear-time lower bounds

Remember the \( \Omega(n \log n) \) lower bound for comparison-based sorting?
Much easier lower bounds exist for many problems!

Linear lower bounds
For any problem with input size \( n \),
where changing any part of the input could change the answer,
any correct algorithm must take \( \Omega(n) \) time.

What does this tell us about integer addition?

Multiplication
Let’s remember how we multiplied multi-digit integers in grade school.
Standard multiplication

def smul(X, Y, B):
    n = len(X)
    A = zero-filled array of length (2*n)
    T = zero-filled array of length n
    for i in range(0, n):
        # set T = X * Y[i]
        carry = 0
        for j in range(0, n):
            T[j] = (X[j] * Y[i] + carry) % B
            carry = (X[j] * Y[i] + carry) // B
        # add T to A, the running sum
        A[i : i+n+1] = add(A[i : i+n], T[0 : n], B)
        A[i+n] += carry
    return A

Divide and Conquer

Maybe a divide-and-conquer approach will yield a faster multiplication algorithm.

Let’s split the digit-lists in half. Let \( m = \left\lfloor \frac{n}{2} \right\rfloor \) and write \( x = x_0 + B^m x_1 \) and \( y = y_0 + B^m y_1 \).
Then we multiply \( xy = x_0 y_0 + x_0 y_1 B^m + x_1 y_0 B^m + x_1 y_1 B^{2m} \).

For example, if \( x = 7407 \) and \( y = 2915 \), then we get:

<table>
<thead>
<tr>
<th>Integers</th>
<th>Array representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 7407</td>
<td>[7, 0, 4, 7]</td>
</tr>
<tr>
<td>y = 2915</td>
<td>[5, 1, 9, 2]</td>
</tr>
<tr>
<td>x0 = 07</td>
<td>X0 = [7, 0]</td>
</tr>
<tr>
<td>x1 = 74</td>
<td>X1 = [4, 7]</td>
</tr>
<tr>
<td>y0 = 15</td>
<td>Y0 = [5, 1]</td>
</tr>
<tr>
<td>y1 = 29</td>
<td>Y1 = [9, 2]</td>
</tr>
</tbody>
</table>

Recurrences for Multiplication

Standard multiplication has running time

\[
T(n) = \begin{cases} 
1, & n = 1 \\
1 + T(n-1), & n \geq 2 
\end{cases}
\]

The divide-and-conquer way has running time

\[
T(n) = \begin{cases} 
1, & n = 1 \\
1 + 4T\left(\frac{n}{2}\right), & n \geq 2 
\end{cases}
\]
Karatsuba’s Algorithm

The equation:

\[(x_0 + x_1 B^m)(y_0 + y_1 B^m) = x_0 y_0 + x_1 y_1 B^{2m} + ((x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1)B^m\]

leads to an algorithm:

1. Compute two sums: \(u = x_0 + x_1\) and \(v = y_0 + y_1\).
2. Compute three \(m\)-digit products: \(x_0 y_0\), \(x_1 y_1\), and \(uv\).
3. Sum them up and multiply by powers of \(B\) to get the answer:

\[xy = x_0 y_0 + x_1 y_1 B^{2m} + (uv - x_0 y_0 - x_1 y_1)B^m\]

Karatsuba Example

\[x = 7407 = 7 + 74*100\]
\[y = 2915 = 15 + 29*100\]

\[u = x_0 + x_1 = 7 + 74 = 81\]
\[v = y_0 + y_1 = 15 + 29 = 44\]

\[x_0 y_0 = 7*15 = 105\]
\[x_1 y_1 = 74*29 = 2146\]
\[u v = 81*44 = 3564\]

\[x y = 105 + 2146*10000 + (3564 - 105 - 2146)*100\]
\[= 105 + 1313\]
\[+ 2146\]
\[= 21591405\]

Analyzing Karatsuba

\[T(n) = \begin{cases} 1, & n \leq 1 \\ n + 3T\left(\frac{n}{2}\right), & n \geq 2 \end{cases}\]

**Crucial difference:** The coefficient of \(T\left(\frac{n}{2}\right)\).
Beyond Karatsuba

**History Lesson:**
- 1962: Karatsuba: $O(n^{1.59})$
- 1963: Toom & Cook: $O(n^{1.47})$, $O(n^{1+\epsilon})$
- 1971: Schönhage & Strassen: $O(n\log n \log \log n)$
- 2007: Fürer: $O(n\log n)^2 \log^* n$

**Lots of work to do in algorithms!**

---

### Recurrences

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Recurrence</th>
<th>Asymptotic big-$\Theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinarySearch</td>
<td>$1 + T(n/2)$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>LinearSearch</td>
<td>$1 + T(n-1)$</td>
<td>$n$</td>
</tr>
<tr>
<td>MergeSort (space)</td>
<td>$n + T(n/2)$</td>
<td>$n$</td>
</tr>
<tr>
<td>MergeSort (time)</td>
<td>$n + 2T(n/2)$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>KaratsubaMul</td>
<td>$n + 3T(n/2)$</td>
<td>$n^{\lg 3}$</td>
</tr>
<tr>
<td>SelectionSort</td>
<td>$n + T(n-1)$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>StandardMul</td>
<td>$n + 4T(n/2)$</td>
<td>$n^2$</td>
</tr>
</tbody>
</table>

---

### Master Method A

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

Write $e = \log_b a = \frac{\lg a}{\lg b}$

Three cases:
- $c = e$. Then $T(n) \in \Theta(n^c \log n^{d+1})$.
- $c < e$. Then $T(n) \in \Theta(n^e) = \Theta(n^{\log_b a})$.
- $c > e$. Then $T(n) \in \Theta(n^c \log n^d)$. 
Master Method B

\[ T(n) = aT(n - b) + n^c \log n^d \]

Two cases:
- \( a = 1 \). Then \( T(n) \in \Theta(n^{c+1} \log n^d) \).
- \( a > 1 \). Then \( T(n) \in \Theta(e^n) \), where \( e \) is the positive constant \( a^{1/b} \).

Matrix Multiplication

Review: **Dimensions** = number of rows and columns.

Multiplication of 4 \( \times \) 3 and 4 \( \times \) 2 matrices:

\[
\begin{bmatrix}
7 & 1 & 2 \\
6 & 2 & 8 \\
9 & 6 & 3 \\
1 & 1 & 4
\end{bmatrix}
\begin{bmatrix}
2 & 0 \\
6 & 3 \\
4 & 3
\end{bmatrix}
= \begin{bmatrix}
28 & 9 \\
56 & 30 \\
66 & 27 \\
24 & 15
\end{bmatrix}
\]

Middle dimensions **must** match.

**Running time:**

Divide and Conquer Matrix Multiplication

\[
\begin{bmatrix}
S & T \\
U & V
\end{bmatrix}
\begin{bmatrix}
W & X \\
Y & Z
\end{bmatrix}
= \begin{bmatrix}
SW + TY & SX + TZ \\
UW + VY & UX + VZ
\end{bmatrix}
\]

**Is this faster?**
Strassen's Algorithm

Step 1: Seven products

\[ P_1 = S(X - Z) \]
\[ P_2 = (S + T)Z \]
\[ P_3 = (U + V)W \]
\[ P_4 = V(Y - W) \]
\[ P_5 = (S + V)(W + Z) \]
\[ P_6 = (T - V)(Y + Z) \]
\[ P_7 = (S - U)(W + X) \]

Step 2: Add and subtract

\[
\begin{bmatrix}
S & T \\
U & V
\end{bmatrix}
\begin{bmatrix}
W & X \\
Y & Z
\end{bmatrix} =
\begin{bmatrix}
P_5 + P_4 - P_2 + P_6 \\
P_3 + P_4 \\
P_1 + P_2
\end{bmatrix}
\begin{bmatrix}
P_5 + P_4 - P_2 + P_6 \\
P_3 + P_4 \\
P_1 + P_5 - P_3 - P_7
\end{bmatrix}
\]

Fibonacci

Here's a basic algorithm to compute \( f_n \):

\[
\text{fib}(n)
\]
Input: Non-negative integer \( n \)
Output: \( f_n \)

\[
def \text{fib}(n):
\quad \text{if } n \leq 1:
\quad \quad \text{return } n
\quad \text{else:}
\quad \quad \text{return } \text{fib}(n-1) + \text{fib}(n-2)
\]

Is this fast?

Recursion tree for \( \text{fib}(6) \)
Memoization
How to avoid repeated, identical function calls?

**Memoization** means saving the results of calls in a table:

```python
def fib_memo(n):
    fib_table = {}  # empty hash table
    if n not in fib_table:
        if n <= 1:
            return n
        else:
            fib_table[n] = fib_memo(n-1) + fib_memo(n-2)
    return fib_table[n]
```

See the original function?

Recursion tree for `fib_memo(6)`

Cost of Memoization

- How should the table $T$ be implemented?

- Analysis
**Matrix Chain Multiplication**

**Problem**
Given $n$ matrices $A_1, A_2, \ldots, A_n$, find the best order of operations to compute the product $A_1 A_2 \cdots A_n$.

Matrix multiplication is associative but not commutative.
In summary: where should we put the parentheses?

**Example**

\[
\begin{bmatrix}
4 & 9 \\
1 & 6 \\
9 & 7 \\
0 & 9 \\
2 & 0 \\
\end{bmatrix}
\times
\begin{bmatrix}
2 & 1 & 5 & 6 & 4 & 5 \\
8 & 0 & 9 & 1 & 8 & 4 \\
\end{bmatrix}
\times
\begin{bmatrix}
6 & 5 & 4 \\
8 & 8 & 5 \\
4 & 4 & 4 \\
0 & 7 & 0 \\
6 & 4 & 2 \\
1 & 7 & 5 \\
\end{bmatrix}
\]

$X \times Y \times Z$
$5 \times 2 \times 2 \times 6 \times 6 \times 3$

**Computing minimal mults**

**Idea:** Figure out the final multiplication, then use recursion to do the rest.

\[
\text{mm(D)}
\]

**Input:** Dimensions array $D$ of length $n+1$

**Output:** Least number of mults to compute the matrix chain product

```python
def mm(D):
    n = len(D) - 1
    if n == 1:
        return 0
    else:
        fewest = float('inf') # (just a placeholder)
        for i in range(1, n):
            t = (mm(D[0 : i+1]) + D[0]*D[i]*D[n] + mm(D[i : n+1]))
            if t < fewest:
                fewest = t
        return fewest
```
Analyzing \( mm(D) \)

\[
T(n) = \begin{cases} 
1, & n = 1 \\
 n + \sum_{i=1}^{n-1} (T(i) + T(n-i)), & n \geq 2 
\end{cases}
\]

Memoized minimal mults

Let's use our **general tool** for avoiding repeated recursive calls:

\( mmm(D) \)

- **Input:** Dimensions array \( D \) of length \( n + 1 \)
- **Output:** Least number of mults to compute the matrix chain product

```python
mm_table = {}
def mmm(D):
    n = len(D) - 1
    if D not in mm_table:
        if n == 1: mm_table[D] = 0
        else:
            fewest = float('inf')
            for i in range(1, n):
                t = ( mmm(D[0 : i+1]) + D[0]*D[i]*D[n] + mmm(D[i : n+1]) )
                if t < fewest: fewest = t
            mm_table[D] = fewest
    return mm_table[D]
```

Analyzing \( mmm(D) \)

- **Cost of each call, not counting recursion:**

- **Total number of recursive calls:**
Problems with Memoization

1. What data structure should $T$ be?
2. Tricky analysis
3. Too much memory?

Solution: Dynamic Programming

- Store the table $T$ explicitly, for a **single problem**
- Fill in the entries of $T$ needed to solve the current problem
- Entries are computed in order so recursion is never required
- Final answer can be looked up in the filled-in table

Dynamic Minimal Mults Example

Multiply $(8 \times 5)$ times $(5 \times 3)$ times $(3 \times 4)$ times $(4 \times 1)$ matrices.

$D = [8, 5, 3, 4, 1], \ n = 4$

Make a table for the value of $mm(D[i..j])$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>60</td>
<td></td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Minimal Mults Algorithm

```python
def dmm(*D):
    n = len(D) - 1
    A = new (n+1) by (n+1) array
    for diag in range(1, n+1):
        for row in range(0, n-diag+1):
            col = diag + row
            # This part is just like the original!
            if diag == 1:
                A[row][col] = 0
            else:
                A[row][col] = float('inf')
                for i in range(row+1, col):
                    if t < A[row][col]:
                        A[row][col] = t
    return A[0][n]
```