Representing Big Integers

**Multiple-precision** integers can’t be stored in a single machine word like an ‘int’.

Why are these important computationally?

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

**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

1. \( \text{carry} := 0 \)
2. \( A := \text{new array of length } n+1 \)
3. \( \text{for } i \text{ from } 0 \text{ to } n-1 \)
4. \( A[i] := (X[i] + Y[i] + \text{carry}) \mod B \)
5. \( \text{carry} := (X[i] + Y[i] + \text{carry}) / B \)
6. \( \text{end for} \)
7. \( A[n] := \text{carry} \)
8. \( \text{return } A \)

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

1 \ A := \text{new array of length (2*n)}
2 \ A := [0 \ 0 \ldots \ 0]
3 \ T := \text{new array of length (n+1)}
4 \ \text{for i from 0 to n-1}
5 \ \text{-- set T to X times Y[i] --}
6 \ \text{carry := 0}
7 \ \text{for j from 0 to n-1}
8 \ T[j] := (X[j] \ast Y[i] + \text{carry}) \mod B
9 \ \text{carry := (X[j] \ast Y[i] + carry) / B}
10 \ \text{end for}
11 \ T[n] := \text{carry}
12 \ \text{-- Add T to A, the running sum --}
13 \ A[i..i+n] := \text{add}(A[i..i+n-1], T[0..n])
14 \ \text{end for}
15 \ \text{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 \( m = \lfloor \frac{n}{2} \rfloor \) 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>X = [7, 0, 4, 7]</td>
</tr>
<tr>
<td>y = 2915</td>
<td>Y = [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

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

The divide-and-conquer way has running time

\[ T(n) = \begin{cases} 
1, & n = 1 \\
 n + 4 T(\frac{n}{2}), & n \geq 2 
\end{cases} \]
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

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

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

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

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

Analyzing Karatsuba

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

Crucial difference: The coefficient of \(T(\frac{n}{2})\).
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-Θ</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>$n^{\log_3 3}$</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 $e = \log_b a = \frac{\log a}{\log b}$

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

\[ T(n) = aT(n - b) + nc \log n \]

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

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

Step 2: Add and subtract

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

Fibonacci

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

\[
\text{fib}(n)
\]

Input: Non-negative integer \( n \)
Output: \( f_n \)

1. if \( n \leq 1 \) then return \( n \)
2. else 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:

```plaintext
fibmemo(n)
Input: Non-negative integer n
Output: fn

1 if T[n] is unset then
2 if n <= 1 then T[n] := n
3 else T[n] := fibmemo(n-1) + fibmemo(n-2)
4 end if
5 return T[n]
```

See the original function?

Recursion tree for fibmemo(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 $A_1A_2\cdots A_n$.

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

Example

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

$X Y Z$

$5 \times 2$

$2 \times 6$

$6 \times 3$

Computing minimal mults

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

```plaintext
mm(D)
Input: Dimensions array D of length $n + 1$
Output: Least number of mults to compute the matrix chain product
1 if n = 1 then return 0
2 else
3 fewest := infinity --(just a placeholder)--
4 for i from 1 to n-1 do
5 t := mm(D[0..i]) + D[0]*D[i]*D[n] + mm(D[i..n])
6 if t < fewest then fewest := t
7 end for
8 return fewest
9 end if
```

Analyzing \( m(D) \)

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

Memoized minimal multis

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

\begin{verbatim}
mmm(D)
Input: Dimensions array D of length \( n + 1 \)
Output: Least number of mults to compute the matrix chain product
1 if T[D] is unset then
2   if n = 1 then T[D] := 0
3 else
4   T[D] := infinity --(just a placeholder)---
5   for i from 1 to n-1 do
6     t := mmm(D[0..i]) + D[0]*D[i]*D[n] + mmm(D[i..n])
7     if t < T[D] then T[D] := t
8   end for
9 end if
10 end if
11 return T[D]
\end{verbatim}

Analyzing \( mmm(D) \)

- Cost of each call, not counting recursion:

- Total number of recursive calls:
Problems with Memoization

- What data structure should $T$ be?
- Tricky analysis
- 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></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

dmm($D$)

Input: Dimensions array $D$ of length $n + 1$
Output: Least number of mults to compute the matrix chain product

```
1 A := new (n+1) by (n+1) array
2 for diag from 1 to n do
3   for row from 0 to (n-diag) do
4     col := diag + row
5     -- This part just like the original version --
6     if diag = 1 then A[row,col] := 0
7     else
9       for i from row+1 to col-1 do
12   end for
13 end if
14 end for
15 return A[0,n]
```