Multiplying Two Numbers

\[ f(n) = \text{number of elementary operations when multiplying two n-digit numbers}. \]

By the naive algorithm, we have

\[ ab = a_1 a_2 10^n + [a_1 b_2 + a_2 b_1] 10^n + a_2 b_2 \]

\[ \Rightarrow \text{breaking the original problem into four subproblems of size } \frac{n}{2} \text{ each.} \]

The addition and digit shift take roughly \( n \) elementary operations

\[ f(n) = 4 f\left(\frac{n}{2}\right) + n \quad (\text{assume } f(1) = 1). \]

\[ \Rightarrow f(n) = 2n^2 - n \quad (\text{check by induction}). \]

\[ \Rightarrow \text{number of elementary operations is roughly } n^2, \text{ which asymptotically is not any better than the grade school algorithm.} \]
Multiplying Two Numbers

Karatsuba Algorithm

Idea: it is enough to have only three recursive calls.

We compute \( a_1 b_1 \), and \( a_2 b_2 \) using recursion, but \( a_1 b_2 + b_1 a_2 \) is computed in one recursive step.

\[
a_1 b_2 + b_1 a_2 = (a_1 + b_2) (b_1 + a_2) - a_1 b_1 - a_2 b_2
\]

Only one multiplication involving two numbers with \( \frac{n}{2} \) digits. In fact, we may have \( \frac{n}{2} + 1 \) digits, but it will not matter.

\[
f(n) = 3 f(\frac{n}{2}) + 2n \quad (f(1) = 1)
\]

\[
\Rightarrow f(n) = 3^{\log_3 n} - 2n
\]

\[
= \text{roughly } 3^{\log_3 n} \text{ elementary operations suffice.}
\]

Comment: A more efficient implementation of Karatsuba is:

\[
a b = (10^\frac{n}{2}) a_1 b_1 - 10^{\frac{n}{2}} (a_1 - a_2) (b_1 - b_2) + (10^{\frac{n}{2} + 1}) a_2 b_2
\]

The solution to \( f(n) \) is computed by recursion trees.
Matrix Multiplication

**Input:** A, B two n x n matrices

**Output:** C = AxB.

First, the standard matrix multiplication costs $n^3$ elementary operations (check!)

**Recursive Algorithm**

Divide A, B into four blocks:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]

\[
C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]

\[
= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}
\]

\[
\Rightarrow \text{This consists of 8 multiplications of pairs of 2x2 submatrices.}\]
Adding the resulting submatrix multiplication costs roughly $n^2$ elementary operations.

\[ f(n) = 8 f\left(\frac{n}{2}\right) + n^2, \quad f(1) = 1. \]

Here, $f(n) \approx n^3$, so this is not any better than naive matrix multiplication.

**But there is an improvement!** Only 7 multiplications suffice. [Not trivial!]

\[ f(n) = 7 f\left(\frac{n}{2}\right) + n^2, \quad f(1) = 1. \]

Here, $f(n) \approx n^{\log_2 7} < n^{\log_2 8} = n^3$.

Solution to $f(n)$ is computed by recursion trees.
Evaluation of polynomials

\[ P(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n. \]

\( a_0, a_1, \ldots, a_n \) are real numbers.

**Goal:** Given \( x = x_0 \), evaluate \( P(x_0) \)

**Simple-minded way:** Proceed over \( n \) iterations, at each iteration \( i \), compute \( a_i x^i = a_i \cdot x^{i-1} \cdot x \). That is, we use the value of \( x^{i-1} \) from previous iteration.

\[ \Rightarrow \text{at each iteration } i \geq 2, \text{we have 2 multiplications to perform: one for } x \text{ and the other for } a_i. \text{ At iteration } i = 1, \text{ we have only one multiplication operation.} \]

\[ \text{Overall, } (2n-1) \text{ operations.} \]

**But we can improve this to only } n \text{ operations:**

Write:

\[ b_n = a_n \]
\[ b_{n-1} = a_{n-1} + b_n x_0 \]
\[ \vdots \]
\[ b_0 = a_0 + b_1 x_0 \]

\[ \Rightarrow b_0 \text{ is the value of } P(x_0). \]
That is, we formed a recursive equation for $b_0$. In fact, this follows from:

$$ P(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x))) \ldots $$

Now, we observe that $b_0$ can be obtained by just $N$ multiplications. Indeed, in order to move from $b_0$ to $b_{n-1}$, we need to make one multiplication, same with the move from $b_{n-1}$ to $b_{n-2}$, from $b_{n-3}$ to $b_{n-4}$, and so on... until the last move $b_1$ to $b_0$, where $b_0$ is the value of $P(x_0)$. 
Searching an element in a list

1, 5, 10, 7, 2, 11, 6

\[ \text{find}(5), \text{find}(7), \text{find}(20) \rightarrow \text{null} \]

We search a given element in the list by traversing all of its elements (say from left to right). If the element is there, we return a pointer; if not, this pointer is null.

If the list consists of \( n \) elements, number of comparisons needed is at most \( n \).

Can we do better?

Suppose next the elements in the list are sorted (say, in an increasing order)

\[ 1, 2, 5, 6, 7, 10, 11. \]

How shall we search now?

Search 7: first compare 6, 7: \( 6 < 7 \)

\[ \Rightarrow \text{continue search at right half} \]

\[ \{7, 10, 11\} \]

- compare 10, 7: \( 7 < 10 \)

\[ \Rightarrow \text{continue search at left half} \]
- Now, we are left with a single element in the list. Compare 7, 7 ⇒ 7 = 7.
  We found the element!

This procedure is called a **binary search**. Let us assume the elements are given in an array, A.

Idea: Compute middle value in A with search key. Based on the result, search the upper or lower half of the array.

Next, we describe the algorithm with a pseudo-code: very similar to a real (C/C++/Pascal) code, but does not suffer from software engineering issues. A compact format of the computational steps of the algorithm.

```
function BinarySearch(key, A, low, high)
    if (low == high) then return (A[low] == key).
    else
        mid = (low + high) / 2.
        if (A[mid] == key) return true
        else
            if (A[mid] > key) return BinarySearch(key, A, low, mid-1)
            else return BinarySearch(key, A, mid+1, high)
```
function BinarySearch(key, A, low, high)

if (low == high) then return (A[low] == key)
else
    mid = (low + high) / 2
    if (A[mid] == key) return true
    else
        if (A[mid] > key) return BinarySearch(key, A, low, mid-1)
        else if (A[mid] < key) return BinarySearch(key, A, mid+1, high)

In the code above, each recursive call halves the array A, and we make exactly one recursive call.

\[ f(n) = \text{number of comparisons made by Binary Search.} \]

\[ \Rightarrow f(n) \leq f\left(\frac{n}{2}\right) + 1 \quad (\text{check!}) \]

\[ f(n) \leq \log_2 n \]

[We have \( \log_2 n \) steps until we reduce \( n \) to 1, in each step we make one comparison.]