More divide & conquer algorithms: fast int/matrix multiplication
Outline

1 Recap

2 Binary search

3 Integer multiplication

4 Fast matrix multiplication (Strassen’s algorithm)
1 Recap

2 Binary search

3 Integer multiplication

4 Fast matrix multiplication (Strassen’s algorithm)
Review of the last lecture

In the last lecture we discussed

- Asymptotic notation \((O, \Omega, \Theta, o, \omega)\)
- The divide & conquer principle
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively.
  - **Combine** the solutions to the subproblems into the solution for the original problem.
- Application: mergesort
- Solving recurrences
mergesort \((A, left, right)\)

if right == left then
    return
end if

mid = left + \lfloor (right - left)/2 \rfloor
mergesort \((A, left, mid)\)
mergesort \((A, mid + 1, right)\)
merge \((A, left, right, mid)\)

- Initial call: `mergesort(A, 1, n)`
- Subroutine `merge` merges two sorted lists of sizes \([n/2]\), \([n/2]\) into one sorted list of size \(n\) in time \(\Theta(n)\).
The running time of **mergesort** satisfies:

\[ T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0 \]

\[ T(1) = c \]

This structure is typical of **recurrence relations**:  
- an *inequality* or *equation* bounds \( T(n) \) in terms of an expression involving \( T(m) \) for \( m < n \)
- a base case generally says that \( T(n) \) is constant for small constant \( n \)

**Remarks**
- We ignore floor and ceiling notations
- A recurrence does **not** provide an asymptotic bound for \( T(n) \): to this end, we must **solve** the recurrence
The technique consists of three steps

1. Analyze the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum over all levels of recursion

Example: analysis of running time of *mergesort*

\[ T(n) = 2T(n/2) + cn, \quad n \geq 2 \]
\[ T(1) = c \]
The running time of many recursive algorithms is given by

\[ T(n) = aT\left(\frac{n}{b}\right) + cn^k, \quad \text{for } a, c > 0, \ b > 1, \ k \geq 0 \]

What is the recursion tree for this recurrence?

- \(a\) is the branching factor
- \(b\) is the factor by which the size of each subproblem shrinks

⇒ at level \(i\), there are \(a^i\) subproblems, each of size \(n/b^i\)

⇒ each subproblem at level \(i\) requires \(c(n/b^i)^k\) work

⇒ the height of the tree is \(\log_b n\) levels

⇒ Total work: \(\sum_{i=0}^{\log_b n} a^i c(n/b^i)^k = cn^k \sum_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i\)
Theorem 1 (Master theorem).

If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ for some constants $a > 0$, $b > 1$, $k \geq 0$, then

\[
T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{, if } a > b^k \\
O(n^k \log n) & \text{, if } a = b^k \\
O(n^k) & \text{, if } a < b^k
\end{cases}
\]

Example: running time of mergesort

- $T(n) = 2T(n/2) + cn$:
  
  $a = 2$, $b = 2$, $k = 1$, $b^k = 2 = a \Rightarrow T(n) = O(n \log n)$
Today

1 Recap

2 Binary search

3 Integer multiplication

4 Fast matrix multiplication (Strassen’s algorithm)
Searching a sorted array

- **Input:**
  1. sorted list $A$ of $n$ integers;
  2. integer $x$

- **Output:**
  - index $j$ such that $1 \leq j \leq n$ and $A[j] = x$; or
  - **no** if $x$ is not in $A$
Searching a sorted array

▶ Input:
  1. sorted list $A$ of $n$ integers;
  2. integer $x$

▶ Output:
  ▶ index $j$ such that $1 \leq j \leq n$ and $A[j] = x$; or
  ▶ no if $x$ is not in $A$

Example: $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$
Searching a sorted array

► **Input:**
   1. sorted list $A$ of $n$ integers;
   2. integer $x$

► **Output:**
   ▶ index $j$ such that $1 \leq j \leq n$ and $A[j] = x$; or
   ▶ no if $x$ is not in $A$

**Example:** $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$

**Idea:** use the fact that the array is sorted and probe specific entries in the array.
First, probe the middle entry. Let $\text{mid} = \lceil n/2 \rceil$.

- If $x == A[\text{mid}]$, return $\text{mid}$.
- If $x < A[\text{mid}]$ then look for $x$ in $A[1, \text{mid} - 1]$;
- Else if $x > A[\text{mid}]$ look for $x$ in $A[\text{mid} + 1, n]$.

Initially, the entire array is “active”, that is, $x$ might be anywhere in the array.

Suppose $x > A[\text{mid}]$.

Then the active area of the array, where $x$ might be, is to the right of $\text{mid}$.
Binary search pseudocode

binarysearch(A, left, right)
    mid = left + ⌈(right − left)/2⌉
    if x == A[mid] then
        return mid
    else if right == left then
        return no
    else if x > A[mid] then
        left = mid + 1
    else
        right = mid − 1
    end if
    binarysearch(A, left, right)

Initial call: binarysearch(A, 1, n)
Observation: At each step there is a region of $A$ where $x$ could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If $n$ is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If $n$ is even, then we are throwing away at least $n/2$ elements.
**Observation:** At each step there is a region of $A$ where $x$ could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If $n$ is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If $n$ is even, then we are throwing away at least $n/2$ elements.

Hence the recurrence for the running time is

$$T(n) \leq T(n/2) + O(1)$$
Here are two ways to argue about the running time:

1. Master theorem: $b = 2, a = 1, k = 0 \Rightarrow T(n) = O(\log n)$.

2. We can reason as follows: starting with an array of size $n$,
   - After $k$ probes, the array has size at most $\frac{n}{2^k}$ (every time we probe an entry, the active portion of the array halves).
   - After $k = \log n$ probes, the array has constant size. We can now search linearly for $x$ in the constant size array.
   - We spend constant work to halve the array (why?). Thus the total work spent is $O(\log n)$. 
Concluding remarks on binary search

1. The right data structure can improve the running time of the algorithm significantly.
   - What if we used a linked list to store the input?
   - Arrays allow for random access of their elements: given an index, we can read any entry in an array in time $O(1)$ (constant time).

2. In general, we obtain running time $O(\log n)$ when the algorithm does a constant amount of work to throw away a constant fraction of the input.
1 Recap

2 Binary search

3 Integer multiplication

4 Fast matrix multiplication (Strassen’s algorithm)
How do we multiply two integers \( x \) and \( y \)?

Elementary school method: compute a partial product by multiplying every digit of \( y \) separately with \( x \) and then add up all the partial products.

Remark: this method works the same in base 10 or base 2.

Examples: \((12)_{10} \cdot (11)_{10}\) and \((1100)_{2} \cdot (1011)_{2}\)

\[
\begin{array}{c}
12 \\
\times 11 \\
\hline
12 \\
+ 12 \\
\hline
132
\end{array}
\quad
\begin{array}{c}
1100 \\
\times 1011 \\
\hline
1100 \\
1100 \\
0000 \\
+ 1100 \\
\hline
10000100
\end{array}
\]
A more reasonable model of computation: a single operation on a pair of digits (bits) is a primitive computational step.

Assume we are multiplying \( n \)-digit (bit) numbers.

- \( O(n) \) time to compute a partial product.
- \( O(n) \) time to combine it in a running sum of all partial products so far.

\[
\Rightarrow \text{There are } n \text{ partial products, each consisting of } n \text{ bits, hence total number of operations is } O(n^2).
\]

*Can we do better?*
Consider \( n \)-digit decimal numbers \( x, y \).

\[
\begin{align*}
    x &= x_n \cdots x_{n/2} x_{n/2-1} \cdots x_0 = x_H \cdot 10^{n/2} + x_L \\
    y &= y_n \cdots y_{n/2} y_{n/2-1} \cdots y_0 = y_H \cdot 10^{n/2} + y_L
\end{align*}
\]

**Idea:** rewrite each number as the sum of the \( n/2 \) high-order digits and the \( n/2 \) low-order digits.

Where each of \( x_H, x_L, y_H, y_L \) is an \( n/2 \)-digit number.
Examples

- $n = 2, x = 12, y = 11$

\[
\begin{align*}
\underbrace{12}_{x} &= \underbrace{1}_{x_H} \cdot \underbrace{10^{1}}_{10^{n/2}} + \underbrace{2}_{x_L} \\
\underbrace{11}_{y} &= \underbrace{1}_{y_H} \cdot \underbrace{10^{1}}_{10^{n/2}} + \underbrace{1}_{y_L}
\end{align*}
\]

- $n = 4, x = 1000, y = 1110$

\[
\begin{align*}
\underbrace{1000}_{x} &= \underbrace{10}_{x_H} \cdot \underbrace{10^{2}}_{10^{n/2}} + \underbrace{0}_{x_L} \\
\underbrace{1110}_{y} &= \underbrace{11}_{y_H} \cdot \underbrace{10^{2}}_{10^{n/2}} + \underbrace{10}_{y_L}
\end{align*}
\]
A first divide & conquer approach

\[ x \cdot y = (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_L) \]
\[ = x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) \cdot 10^{n/2} + x_L y_L \]

In words, we reduced the problem of solving 1 instance of size \( n \) (i.e., one multiplication between two \( n \)-digit numbers) to the problem of solving 4 instances, each of size \( n/2 \) (i.e., computing the products \( x_H y_H, x_H y_L, x_L y_H \) and \( x_L y_L \)).
A first divide & conquer approach

\[
x \cdot y = (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_L) \\
= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) \cdot 10^{n/2} + x_L y_L
\]

In words, we reduced the problem of solving 1 instance of size \( n \) (i.e., one multiplication between two \( n \)-digit numbers) to the problem of solving 4 instances, each of size \( n/2 \) (i.e., computing the products \( x_H y_H, x_H y_L, x_L y_H \) and \( x_L y_L \)).

This is a divide and conquer solution!

- Recursively solve the 4 subproblems.
- Multiplication by \( 10^n \) is easy (shifting): \( O(n) \) time.
- Combine the solutions from the 4 subproblems to an overall solution using 3 additions on \( O(n) \)-digit numbers: \( O(n) \) time.
Running time: $T(n) \leq 4T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- no improvement
Karatsuba’s observation

Running time: $T(n) \leq 4T(n/2) + cn$
- by the Master Theorem: $T(n) = O(n^2)$
- no improvement

However, if we only needed three $n/2$-digit multiplications, then by the Master theorem

$$T(n) \leq 3T(n/2) + cn = O(n^{1.59}) = o(n^2).$$
Karatsuba’s observation

Running time: \( T(n) \leq 4T(n/2) + cn \)
- by the Master Theorem: \( T(n) = O(n^2) \)
- no improvement

However, if we only needed three \( n/2 \)-digit multiplications, then by the Master theorem

\[
T(n) \leq 3T(n/2) + cn = O(n^{1.59}) = o(n^2).
\]

Recall that

\[
x \cdot y = x_H y_H \cdot 10^n + (x_H y_L + x_L y_H)10^{n/2} + x_L y_L
\]

Key observation: we do not need each of \( x_H y_L, x_L y_H \). We only need their sum, \( x_H y_L + x_L y_H \).
A similar situation: multiply two complex numbers $a + bi, c + di$

$$(a + bi)(c + di) = ac + (ad + bc)i + bdi^2$$
Gauss’s observation on multiplying complex numbers

A similar situation: multiply two complex numbers $a + bi, c + di$

$$(a + bi)(c + di) = ac + (ad + bc)i + bdi^2$$

*Gauss’s observation:* can be done with just 3 multiplications

$$(a + bi)(c + di) = ac + ((a + b)(c + d) - ac - bd)i + bdi^2,$$

at the cost of few extra additions and subtractions.

* Unlike multiplications, additions and subtractions of $n$-digit numbers are cheap: $O(n)$ time!
Karatsuba’s algorithm

\[ x \cdot y = (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_H) = x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L \]

Similarly to Gauss’s method for multiplying two complex numbers, compute only the three products

\[ x_H y_H, \; x_L y_L, \; (x_H + x_L)(y_H + y_L) \]

and obtain the sum \( x_H y_L + x_L y_H \) from

\[ (x_H + x_L)(y_H + y_L) - x_H y_H - x_L y_L = x_H y_L + x_L y_H. \]

Combining requires \( O(n) \) time hence

\[ T(n) \leq 3T(n/2) + cn = O(n^{\log_2 3}) = O(n^{1.59}) \]
Let $k$ be a small constant.

**Integer-Multiply** $(x, y)$

```plaintext
if $n == k$ then
    return $xy$
end if
write $x = x_H 10^{n/2} + x_L, y = y_H 10^{n/2} + y_L$
compute $x_H + x_L, y_H + y_L$
product = Integer-Multiply$(x_H + x_L, y_H + y_L)$
$x_H y_H = Integer-Multiply(x_H, y_H)$
$x_L y_L = Integer-Multiply(x_L, y_L)$
return $x_H y_H 10^n + (product - x_H y_H - x_L y_L)10^{n/2} + x_L y_L$
```
Concluding remarks

- To reduce the number of multiplications we do few more additions/subtractions: these are fast compared to multiplications.
- There is no reason to continue with recursion once $n$ is small enough: the conventional algorithm is probably more efficient since it uses fewer additions.
- When we recursively compute $(x_H + x_L)(y_H + y_L)$, each of $x_H + x_L$, $y_H + y_L$ might be $(n/2 + 1)$-digit integers. This does not affect the asymptotics.
Today

1 Recap

2 Binary search

3 Integer multiplication

4 Fast matrix multiplication (Strassen’s algorithm)
Matrix multiplication: a fundamental primitive in numerical linear algebra, scientific computing, machine learning and large-scale data analysis.

- Input: $m \times n$ matrix $A$, $n \times p$ matrix $B$
- Output: $m \times p$ matrix $C = AB$

Example: $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $C = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

Lower bounds on matrix multiplication algorithms for $m, p = \Theta(n)$?
for $1 \leq i \leq m$ do
    for $1 \leq j \leq p$ do
        $c_{i,j} = 0$
        for $1 \leq k \leq n$ do
            $c_{i,j} += a_{i,k} \cdot b_{k,j}$
        end for
    end for
end for

Running time?
Can we do better?
A first divide & conquer approach: 8 subproblems

Assume square $A, B$ where $n = 2^k$ for some $k > 0$.

**Idea:** express $A, B$ as $2 \times 2$ block matrices and use the conventional algorithm to multiply the two block matrices.

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

where

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21}
\]

\[
C_{12} = A_{11}B_{12} + A_{12}B_{22}
\]

\[
C_{21} = A_{21}B_{11} + A_{22}B_{21}
\]

\[
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]

**Running time?**
Compute the following ten $n/2 \times n/2$ matrices.

1. $S_1 = B_{11} - B_{22}$
2. $S_2 = A_{11} + A_{12}$
3. $S_3 = A_{21} + A_{22}$
4. $S_4 = B_{21} - B_{11}$
5. $S_5 = A_{11} + A_{22}$
6. $S_6 = B_{11} + B_{22}$
7. $S_7 = A_{12} - A_{22}$
8. $S_8 = B_{21} + B_{22}$
9. $S_9 = A_{11} - A_{21}$
10. $S_{10} = B_{11} + B_{12}$

Running time?
Compute the following seven products of $n/2 \times n/2$ matrices.

1. $P_1 = A_{11}S_1$
2. $P_2 = S_2B_{22}$
3. $P_3 = S_3B_{11}$
4. $P_4 = A_{22}S_4$
5. $P_5 = S_5S_6$
6. $P_6 = S_7S_8$
7. $P_7 = S_9S_{10}$

Compute $C$ as follows:

1. $C_{11} = P_4 + P_5 + P_6 - P_2$
2. $C_{12} = P_1 + P_2$
3. $C_{21} = P_3 + P_4$
4. $C_{22} = P_1 + P_5 - P_3 - P_7$

Running time?
Strassen’s running time and concluding remarks

- Recurrence: \( T(n) = 7T(n/2) + cn^2 \)
- By the Master theorem:
  \[
  T(n) = O(n^{\log_2 7}) = O(n^{2.81})
  \]
- Recently, there is renewed interest in Strassen’s algorithm for **high-performance computing**: thanks to its lower communication cost (number of bits exchanged between machines in the network or data center), it is better suited than the traditional algorithm for multi-core processors.