Efficient Algorithms and Problem Complexity
– Divide and Conquer –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. What is a Divide-and-Conquer Algorithm?

2. Example 1: Mergesort

3. Example 2: Matrix Multiplication
What is a Divide-and-Conquer Algorithm?

Divide et impera (divide and rule)

Historically

A quote attributed to Julius Caesar describing the political (social, ...) strategy to divide those you want to keep under control into competing groups having roughly the same power, so that none of them can become a leader with the power to threaten you.

In computer science

The problem solving strategy that consists in dividing an input into smaller ones, solving them recursively, and combining the solutions to a solution for the original input.
What is a Divide-and-Conquer Algorithm?

General pattern (1)

For a problem instance (input) $I$ is of size $n$ ...

- divide $I$ into $a$ smaller instances $I_1, \ldots, I_a$,
- solve $I_1, \ldots, I_a$, yielding results $R_1, \ldots, R_a$, and
- combine $R_1, \ldots, R_a$ to a result $R$ for $I$.

Typically,...

- $a$ is a constant $\geq 2$
- the size of each $I_j$ is $\leq \lceil n/b \rceil$ for a constant $b > 1$, and
- “divide” and “combine” take $O(n^k)$ steps for a constant $k$. 

What is a Divide-and-Conquer Algorithm?

**General pattern (2)**

Typically,…

- $a$ is a constant $\geq 2$
- the size of each $I_j$ is $\leq \lceil n/b \rceil$ for a constant $b > 1$, and
- “divide” and “combine” take $O(n^k)$ steps for a constant $k$.

⇒ The running time is bounded by the recurrence

$$T(n) \leq aT(n/b) + O(n^k).$$

⇒ We can apply the **Main Recurrence Theorem** to bound $T(n)$
e.g., $a = b = k = 2$ yields the bound $O(n^2)$. 
Recalling Mergesort (1)

- Mergesort sorts an array of items according to their keys.
- We assume that items only consist of keys, that are integers.
- Sorting an array $a$ of size $n > 1$ works by
  1. recursively sorting $a[1, \ldots, \lceil n/2 \rceil]$ and $a[\lceil n/2 \rceil + 1, \ldots, n]$ and
  2. merging the two (now sorted) sub-arrays into the final result.
Recalling Mergesort (2)

The pseudocode:

\[
\text{Mergesort}(a[1, \ldots, n], i, j) \text{ where initially } i = 1, j = n \\
\text{if } i < j \text{ then} \\
\quad k \leftarrow \lfloor (i + j)/2 \rfloor \\
\quad \text{Mergesort}(a, i, k) \\
\quad \text{Mergesort}(a, k + 1, j) \\
\quad \text{Merge}(a, i, k + 1, j)
\]

The obvious implementation of \text{Merge}(a, i, k, j) runs in time \Theta(i - j).
Example 1: Mergesort

Overall time required by Mergesort

For array size $n$

- two recursive calls are executed ($a = 2$),
- each with a problem size $\leq \lceil n/2 \rceil$ ($b = 2$), and
- the time used by the non-recursive part is $\Theta(n)$ ($k = 1$).

$\Rightarrow$ the resulting recurrence relation is

$$T(n) \leq 2T(n/2) + O(n).$$

$\Rightarrow$ Mergesort runs in time $\Theta(n \log n)$
(by the Main Recurrence Theorem, and since $a = 2 = 2^1 = b^k$).
Example 2: Matrix Multiplication

Multiplying two $n \times n$ matrices

Given:

$$A = (a_{ij}) = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \quad \text{and} \quad B = (b_{ij}) = \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix}.$$ 

Task

Compute the product $C = AB$, i.e., $C = (c_{ij})$ with

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$ 

- The obvious algorithm computes the entries $c_{ij}$ one by one.
- The computation of each $c_{ij}$ requires $2n - 1$ arithmetic operations.

$\Rightarrow$ in total, $\Theta(n^3)$ arithmetic operations are used.
Example 2: Matrix Multiplication

Matrices of matrices . . .

How can we do this by divide-and-conquer?

Suppose for simplicity that \( n \) is a power of 2.

\[ \Rightarrow \text{We can write } A, B, \text{ and } C \text{ as } 2 \times 2 \text{ matrices of } n/2 \times n/2 \text{ matrices:} \]

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

Then, \( C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} \) (just the ordinary matrix multiplication, but now with matrices as entries). [Verify!]

\[ \Rightarrow \text{we get a recursive algorithm for matrix multiplication.} \]
A first recursive algorithm

\[
\text{RecMMult}(A, B, C) \text{ where } A, B, C \text{ are } n \times n \text{ matrices, } n = 2^m
\]

if \( n = 1 \) then
  return \( C = (a_{11} b_{11}) \)
else
  for all \( (i, j) \in \{1, 2\}^2 \) do
    \( C_{i,j} = \text{RecMMult}(A_{i1}, B_{1,j}) + \text{RecMMult}(A_{i2}, B_{2,j}) \)
  return \( C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \)

Resulting recurrence relation: \( T(n) \leq 8T(n/2) + O(n^2) \).

\[ \Rightarrow \text{running time } O(n^{\log 8}) = O(n^3). \] :(

The problem is the factor 8!
Strassen’s recursive algorithm

How can we reduce the number of recursive calls?

Strassen’s observation:

If we let

\[
\begin{align*}
D_1 & = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
D_2 & = (A_{21} + A_{22}) \cdot B_{11} \\
D_3 & = A_{11} \cdot (B_{12} - B_{22}) \\
D_4 & = A_{22} \cdot (B_{21} - B_{11}) \\
D_5 & = (A_{11} + A_{12}) \cdot B_{22} \\
D_6 & = (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
D_7 & = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\end{align*}
\]

then

\[
C = \begin{pmatrix}
D_1 + D_4 - D_5 + D_7 & D_3 + D_5 \\
D_2 + D_4 & D_1 - D_2 + D_3 + D_6
\end{pmatrix}.
\]
Running time of Strassen’s algorithm

What do we lose/gain?

- The non-recursive part still requires $O(n^2)$ operations. (It’s a fixed number of matrix additions – but notice that the factor is 18 instead of 4!)
- We need only 7 recursive calls.
- The size of matrices in recursive calls is the same as before.

$\Rightarrow$ the recurrence relation turns into $T(n) \leq 7T(n/2) + O(n^2)$.

$\Rightarrow$ the time required is $O(n^{\lg 7}) = O(n^{2.81\ldots})$. 

Frank Drewes (Umeå University) Efficient Algorithms and Problem Complexity Lecture 2 13 / 14
Concluding notes

Notes

- Strassen’s algorithm beats the naive one only for very large matrices ⇒ in practice, we need to stop the recursion long before $n = 1$.
- The assumption that $n$ is a power of 2 must be removed (e.g., by filling up with zeroes).
- The best known algorithm [Coppersmith/Winograd 1987] solves the problem in time $\Theta(n^{2.376})$ (though with a huge constant factor).
- This is surprisingly near the trivial lower bound $\Omega(n^2)$.

Read Chapter 5 of the textbook, in particular Section 5.3.