

Algorithms and Data Structures

A10. Runtime Analysis: Divide-and-Conquer Algorithms

Gabriele Röger

University of Basel

March 14, 2024

Algorithms and Data Structures

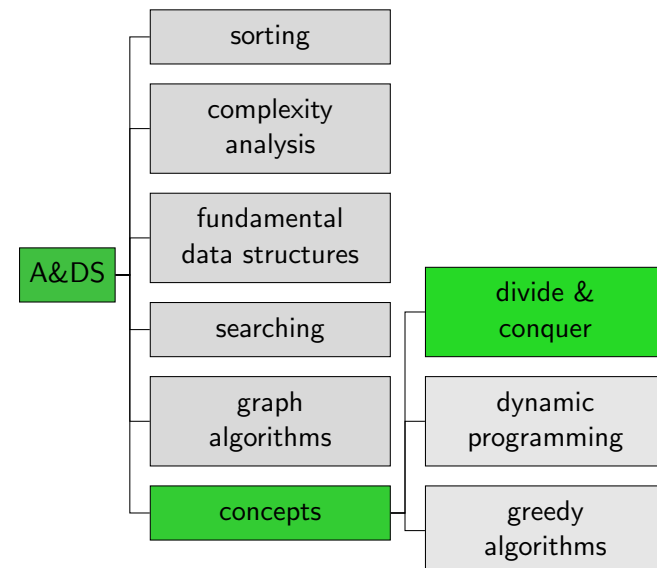
March 14, 2024 — A10. Runtime Analysis: Divide-and-Conquer Algorithms

A10.1 Divide-and-Conquer Algorithms

A10.2 Recurrences

A10.1 Divide-and-Conquer Algorithms

Content of the Course



Recap: Merge Sort

Sort input range with n elements:

- ▶ $n \leq 1$: nothing to do
- ▶ $n > 1$: proceed as follows:

Divide the range into two roughly equally-sized ranges.
Conquer each of them by recursively sorting them.
Combine the sorted subranges to a fully sorted range.

Divide-and-Conquer Algorithm Scheme

Base case: If the problem is small enough, solve it directly without recursing.

Recursive case: Otherwise

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

Example: Multiplication of Square Matrices

$$\text{Square matrix } A_{n \times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Let A, B, C be $n \times n$ matrices. We want to compute $C + A \cdot B$.

For $i, j \in \{1, \dots, n\}$: Update c_{ij} to $c_{ij} + \sum_{k=1}^n a_{ik} \cdot b_{kj}$.

Example: Multiplication of Square Matrices

Direct Computation

```

1 def matrix_multiply(A, B, C, n):
2     for i in range(1, n+1): # i = 1, ..., n
3         for j in range(1, n+1): # j = 1, ..., n
4             for k in range(1, n+1): # k = 1, ..., n
5                 C[i][j] += A[i][k] * B[k][j]

```

Running time $\Theta(n^3)$

Example: Multiplication of Square Matrices

A Simple Divide-and-Conquer Algorithm

Assumption: $n = 2^k$ for some $k \in \mathbb{N}$.

Idea: Divide each matrix into four $n/2 \times n/2$ matrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Can compute $C = A \cdot B$ as

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{bmatrix}$$

Eight $n/2 \times n/2$ multiplications and four $n/2 \times n/2$ additions

Example: Multiplication of Square Matrices

A Simple Divide-and-Conquer Algorithm

function MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)

if $n == 1$ **then**

$$c_{11} = c_{11} + a_{11} \cdot b_{11}$$

return

partition $A, B,$ and C into $n/2 \times n/2$ submatrices

$$A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, \dots, B_{22}, C_{11}, \dots, C_{22}$$

(details omitted; takes constant time)

MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{11}, C_{11}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{12}, C_{12}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{11}, C_{21}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{12}, C_{22}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{21}, C_{11}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{22}, C_{12}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{21}, C_{21}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{22}, C_{22}, n/2$)

Example: Multiplication of Square Matrices

Strassen's Algorithm

- ▶ The previous algorithm still has running time $\Theta(n^3)$.
- ▶ Strassen's algorithm is similar but uses only 7 recursive calls.
- ▶ Idea (with scalars): Compute $x^2 + y^2$ as $(x + y)(x - y)$ with 2 additions, 1 multiplication instead of 2 multiplications, 1 addition
- ▶ Computes the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ with four steps (next slide).

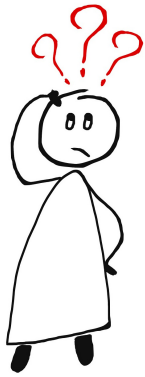
Example: Multiplication of Square Matrices

Strassen's Algorithm (Sketch)

- 1 If n is 1, proceeds as in MATRIX-MULTIPLY-RECURSIVE, otherwise, partition matrices A, B, C as in MATRIX-MULTIPLY-RECURSIVE. This takes $\Theta(1)$ time.
- 2 Create $n/2 \times n/2$ matrices S_1, S_2, \dots, S_{10} , each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices P_1, \dots, P_7 to hold seven matrix products (next step). All 17 matrices can be created/initialized in $\Theta(n^2)$ time.
- 3 Recursively compute each of the seven products P_1, \dots, P_7 .
- 4 Update the four submatrices C_{11}, \dots, C_{22} by adding or subtracting various P_i matrices. This takes $\Theta(n^2)$ time.

Running time $\Theta(n^{\lg 7})$ (with $\lg 7 \approx 2.8073549 < 3$)

Questions

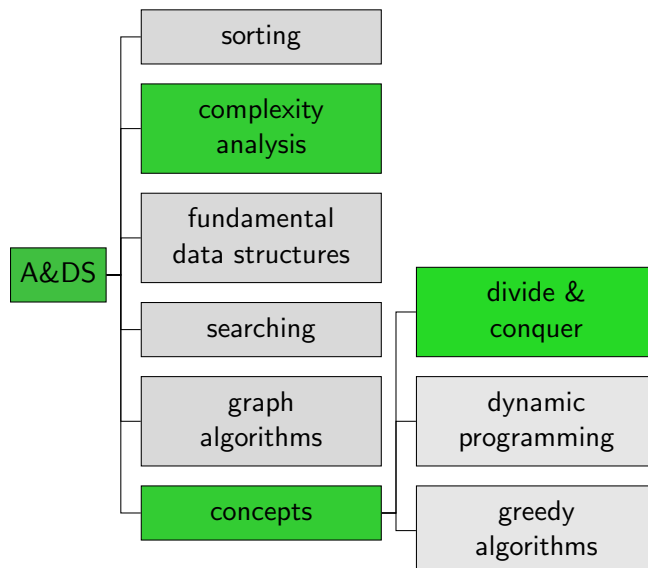


Your Questions?

How can we analyze the running time of such algorithms?

A10.2 Recurrences

Content of the Course



Recurrences

A **recurrence** is a recursively defined function $f : \mathbb{N}_0 \rightarrow \mathbb{R}$ where for almost all n , the value $f(n)$ is defined in terms of the values $f(m)$ for $m < n$.

Example (Fibonacci Series)

$$F(0) = 0 \quad \text{(1st base case)}$$

$$F(1) = 1 \quad \text{(2nd base case)}$$

$$F(n) = F(n-2) + F(n-1) \text{ for all } n \geq 2 \quad \text{(recursive case)}$$

Recurrences occur naturally for the running time of divide-and-conquer algorithms.

Example: Top-Down Merge Sort

```

1 def sort(array):
2     tmp = [0] * len(array) # [0,...,0] with same size as array
3     sort_aux(array, tmp, 0, len(array) - 1)
4
5 def sort_aux(array, tmp, lo, hi):
6     if hi <= lo:
7         return
8     mid = lo + (hi - lo) // 2
9     sort_aux(array, tmp, lo, mid)
10    sort_aux(array, tmp, mid + 1, hi)
11    merge(array, tmp, lo, mid, hi)

```

Analysis for $m = hi - lo + 1$

c_0 for lines 6–7

c_1 for lines 6–8

$c_2 m$ for merge step (takes linear time)

Example: Top-Down Merge Sort

Assumption: $n = 2^k$ for some $k \in \mathbb{N}$

Running time `sort_aux`

▶ $T(1) = c_0$

▶ $T(m) = c_1 + 2T(m/2) + c_2 m$

Example: Multiplication of Square Matrices

A Simple Divide-and-Conquer Algorithm

function MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)

if $n == 1$ **then**

$$c_{11} = c_{11} + a_{11} \cdot b_{11}$$

return

partition A, B , and C into $n/2 \times n/2$ submatrices

$$A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, \dots, B_{22}, C_{11}, \dots, C_{22}$$

(details omitted; takes constant time)

MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{11}, C_{11}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{11}, B_{12}, C_{12}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{11}, C_{21}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{21}, B_{12}, C_{22}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{21}, C_{11}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{12}, B_{22}, C_{12}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{21}, C_{21}, n/2$)

MATRIX-MULTIPLY-RECURSIVE($A_{22}, B_{22}, C_{22}, n/2$)

Example: Multiplication of Square Matrices

A Simple Divide-and-Conquer Algorithm

Assumptions:

▶ $n = 2^k$ for some $k \in \mathbb{N}$,

▶ c_0 is the running time in case $n = 1$, and

▶ c_1 is the time for the partition into submatrices.

Specify a recurrence for the running time $T(n)$ of the algorithm.

Solution:

$$T(1) = c_0$$

$$T(n) = c_1 + 8T(n/2) \quad \text{for } n > 1$$



Algorithmic Recurrences

A recurrence $T(n)$ is **algorithmic** if, for every sufficiently large $n_0 > 0$, the following two properties hold:

- 1 For all $n < n_0$, we have $T(n) = \Theta(1)$.
- 2 For all $n \geq n_0$, every path of recursion terminates in a defined base case within a finite number of recursive invocations.

Convention

- ▶ Whenever a recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic.
- ▶ For non-recursive aspects, we use $\Theta(\cdot)$ (or $O(\cdot)$ if only interested in upper bound).

Examples:

- ▶ $T(m) = 2T(m/2) + \Theta(m)$
for merge sort.
- ▶ $T(n) = 8T(n/2) + \Theta(1)$
for simple recursive matrix multiplication.

Example: Multiplication of Square Matrices

Strassen's Algorithm (Sketch)

- 1 If n is 1, proceeds as in MATRIX-MULTIPLY-RECURSIVE, otherwise, partition matrices A, B, C as in MATRIX-MULTIPLY-RECURSIVE. This takes $\Theta(1)$ time.
- 2 Create $n/2 \times n/2$ matrices S_1, S_2, \dots, S_{10} , each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices P_1, \dots, P_7 to hold seven matrix products (next step). All 17 matrices can be created/initialized in $\Theta(n^2)$ time.
- 3 Recursively compute each of the seven products P_1, \dots, P_7 .
- 4 Update the four submatrices C_{11}, \dots, C_{22} by adding or subtracting various P_i matrices. This takes $\Theta(n^2)$ time.

$$T(n) = \Theta(1) + \Theta(n^2) + 7T(n/2) + \Theta(n^2) = 7T(n/2) + \Theta(n^2)$$

Summary

- ▶ **Divide-and-conquer algorithms** divide the problem into smaller problems of the same kind, solve them (typically recursively) and combine their solution into a solution of the full problem.
- ▶ Their running time can often easily be described with a recurrence.