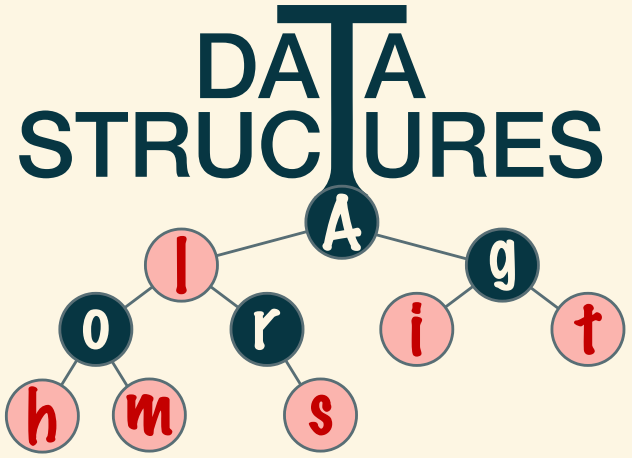
Home



WEEK 02: RUNNING TIMES (LAB)

Michael T. Gastner

2026-01-15

Home

INTRO

Home

LEARNING OBJECTIVES

By the end of this lab, you should be able to:

1. Apply asymptotic notation and the Master Theorem to analyze divide-and-conquer algorithms.
2. Implement divide-and-conquer algorithms as recursive functions, handling base cases and subproblem decomposition.
3. Explain why two algorithms with the same asymptotic complexity can have different real-world performance.
4. Analyze running-time measurements to verify theoretical predictions.

Home

OVERVIEW OF LAB ACTIVITIES

Submit your work for the following activities to xSITe and/or Gradescope (as specified on each slide):

1. Multiple-Choice Questions:
   Answer four questions on asymptotic notation and the Master Theorem.
2. Triple-Loop Matrix Multiplication:
   Implement the straightforward i-j-k loop algorithm.
3. 8-Fold Block-Recursive Matrix Multiplication:
   Implement the divide-and-conquer algorithm.
4. Benchmarking and Analysis:
   Measure running times and compare with theoretical predictions.

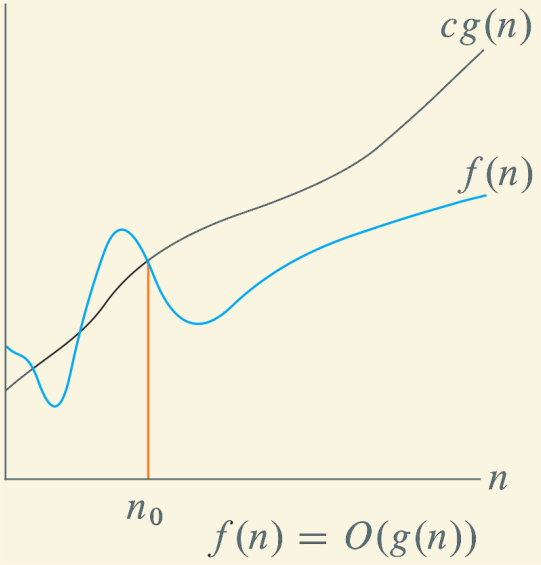
Home

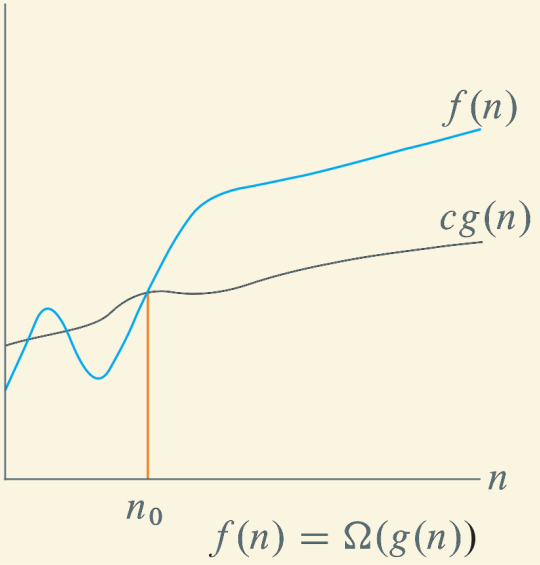
RECAP: O, Ω, AND Θ NOTATION

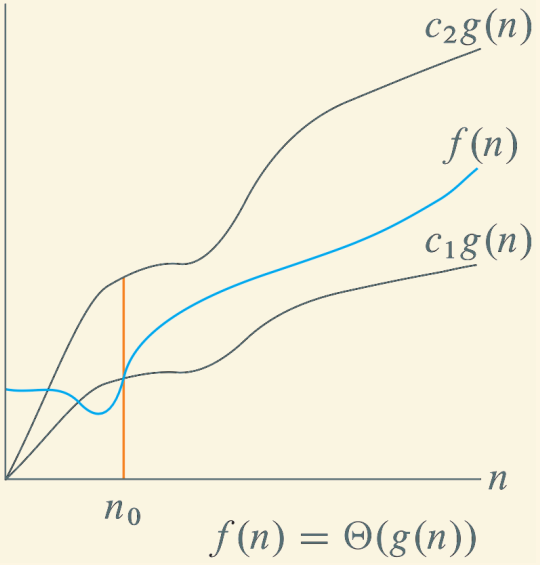
The lecture introduced asymptotic notation to describe the growth of a function $f(n)$:

- $O(g(n))$: upper bound (grows no faster than)
- $\Omega(g(n))$: lower bound (grows at least as fast as)
- $\Theta(g(n))$: tight bound (same order of growth)

The picture below summarizes the idea: for sufficiently large n, $f(n)$ is above, below, or between multiples of $g(n)$.







Home

MULTIPLE-CHOICE QUESTIONS

Home

MULTIPLE-CHOICE QUESTIONS

On the xSITe course page, navigate to **Assessments → Quizzes → Week 02 Lab Exercise 1: Multiple Choice Questions — Running Times of Algorithms**.

Select the best answer for each of the following four questions. You may refer to your notes or the lecture slides.

Home

EXERCISE 1: QUESTION 1

Suppose an algorithm allocates an array a of length n, where each element is a 32-bit integer. The algorithm computes the following sum using a nested loop:

$$S = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a[i] \cdot a[j]$$

Which statement best aligns with the standard definition of **basic steps** in asymptotic analysis?

a. Each multiplication $a[i] \cdot a[j]$ is a basic step only when $i = j$.
b. Each multiplication $a[i] \cdot a[j]$ is *not* a basic step because there are n^2 total multiplications.
c. Each multiplication $a[i] \cdot a[j]$ is a basic step because both operands are fixed-width 32-bit integers.
d. Whether $a[i] \cdot a[j]$ is a basic step depends on whether n is a power of 2.

Home

EXERCISE 1: QUESTION 2

Consider the following pseudocode, where a is an array of length n:

```
for i ← 0 .. n - 1 (both end points inclusive)
    if i > 0 then
        for j ← 0 .. i - 1
            a[i] ← i - j
```

Exactly how many times is the assignment to `a[i]` executed, as a function of n?

a. $n(n+1)/2$
b. n^2
c. $n(n-1)/2$
d. n

Home

EXERCISE 1: QUESTION 3

For sufficiently large n, which ordering from **slowest-growing** to **fastest-growing** is correct?

a. $O(\log^2 n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n)$

b. $O(\log^2 n) \subset O(n) \subset O(\sqrt{n}) \subset O(n \log n)$

c. $O(\sqrt{n}) \subset O(\log^2 n) \subset O(n) \subset O(n \log n)$

d. $O(\log^2 n) \subset O(\sqrt{n}) \subset O(n \log n) \subset O(n)$

Home

EXERCISE 1: QUESTION 4

Use the Master Theorem to solve the recurrence $T(n) = 3T(n/2) + n$. What is the asymptotic solution?

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a function with $f(n) \geq 0$ for all sufficiently large n. Consider recurrences of the form $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ for values of n for which the recurrence is defined (e.g., $n = b^k$). Let $p = \log_b a$.

1. If $f(n) \in O\left(n^{p-\varepsilon}\right)$ for some $\varepsilon > 0$, then $T(n) \in \Theta\left(n^p\right)$.
2. If there exists $k \geq 0$ such that $f(n) \in \Theta\left(n^p \log^k n\right)$, then $T(n) \in \Theta\left(n^p \log^{k+1} n\right)$.
3. If $f(n) \in \Omega\left(n^{p+\varepsilon}\right)$ for some $\varepsilon > 0$ and there exist constants $c < 1$ and n_0 such that for all $n \geq n_0$ we have $a\,f\left(\frac{n}{b}\right) \leq c\,f(n)$, then $T(n) \in \Theta\left(f(n)\right)$.

a. $\Theta(n^2)$ b. $\Theta(n)$ c. $\Theta(n \log n)$ d. $\Theta(n^{\log_2 3})$

Home

MATRIX MULTIPLICATION

Home

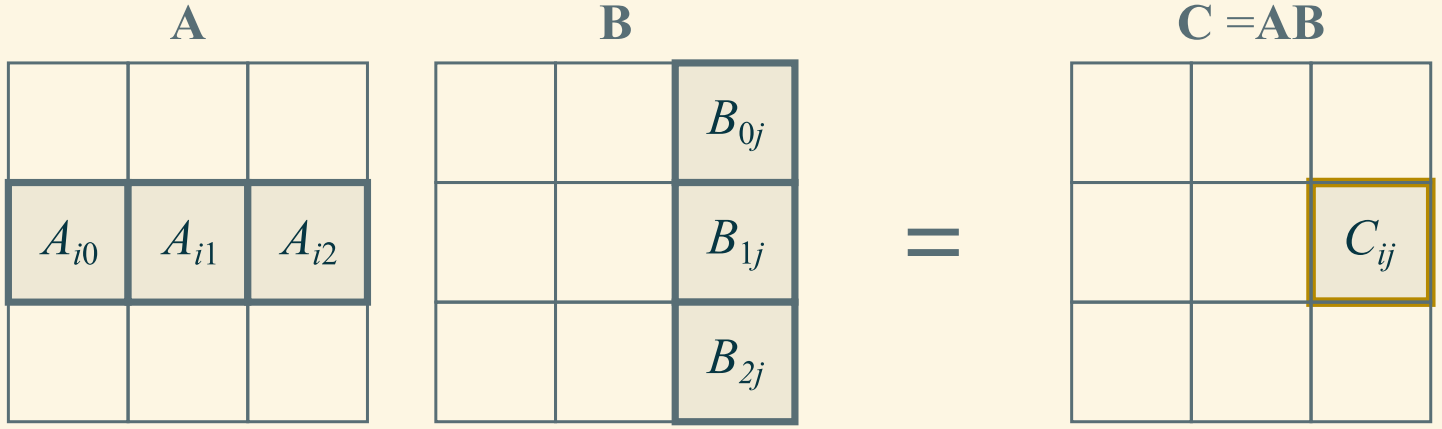
MATRIX MULTIPLICATION: PROBLEM STATEMENT

We study the running time of algorithms that multiply two square matrices.

Given two $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$, the product matrix $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ has elements defined by

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}.$$

Hence, each element C_{ij} is the dot product of $\mathbf{A}$'s i-th row with $\mathbf{B}$'s j-th column. Note that we adopt 0-based indexing.



Home

INPUT FORMAT FOR MATRICES

You can test your code by providing input matrices from a text file with the following format:

- The first line contains a single integer n, representing the number of rows/columns.
- The file then lists $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}_{\text{expected}} = \mathbf{A} \cdot \mathbf{B}$ (each n rows); numbers in a row are comma-separated.
- Blank lines are ignored, and # starts a comment.

Example ($n = 2$):

```
2

# A
1,2
3,4

# B
5,6
7,8

# C_expected = A * B
19,22
43,50
```

Home

C++ IMPLEMENTATION OVERVIEW

Download the starter code from GitHub Pages. The ZIP archive contains a C++ project for the exercises on the following slides:

`main_*.cpp`

Entry points for programs that perform specific tasks, such as correctness checks and benchmarking

`matrix_utilities.cpp` and `matrix_utilities.h`

Matrix types and helper functions (including input parsing)

`matmul_*.cpp` and `matmul_*.h`

Stubs for matrix-multiplication algorithms

`vega_lite_plot.cpp` and `vega_lite_plot.h`

Utilities for plotting

`makefile`

File to compile the project using `make`

Home

UNDERSTANDING THE MATRIX TYPE

The starter code defines the matrix type in `matrix_utilities.h`:

```cpp
using MatrixInt64 = std::vector<std::vector<std::int64_t>>;
```

- `MatrixInt64` is a **type alias** for a 2D vector of 64-bit signed integers.
- Access element (i, j) as `matrix[i][j]`.
- Create an $n \times n$ zero matrix:

```cpp
MatrixInt64 matrix_c(n, std::vector<std::int64_t>(n, 0));
```

Why `int64_t`? When multiplying large matrices, intermediate sums can overflow 32-bit integers (even when the input entries fit in 32-bit). Using 64-bit integers provides a safety margin.

Home

TRIPLE LOOP

Home

EXERCISE 2A: TRIPLE-LOOP MATRIX MULTIPLICATION

Recall the definition of matrix multiplication: $C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$. Your task is to implement this summation as a triple loop over i, j, and k (in this order). In the provided ZIP archive, complete the function stub `matmulIJK()` in `matmul_ijk.cpp`.

To compile and test locally, run the following commands:

```
make clean
make check_ijk
make run TARGET=check_ijk FILE=test_0.txt
```

Upload your solution to Gradescope. Detailed instructions are provided on xSITe (Dropbox → Week 02 Lab Exercise 2a: Gradescope—Triple-Loop Matrix Multiplication).

Home

DIVIDE AND CONQUER

Home

8-FOLD BLOCK-RECURSIVE MATRIX MULTIPLICATION: KEY IDEA

Block-recursive matrix multiplication is a divide-and-conquer algorithm that multiplies matrices by splitting them into equally-sized quadrants:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}.$$

Then compute the four blocks of $\mathbf{C}$ using **8 smaller multiplications** (plus 4 additions):

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\ \mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\ \mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{aligned}$$

Home

DIVIDE AND CONQUER: WHY POWER-OF-2 SIZES?

The recursive algorithm splits matrices into **equal quadrants**. This procedure requires the matrix dimension to be divisible by 2 at each level of recursion.

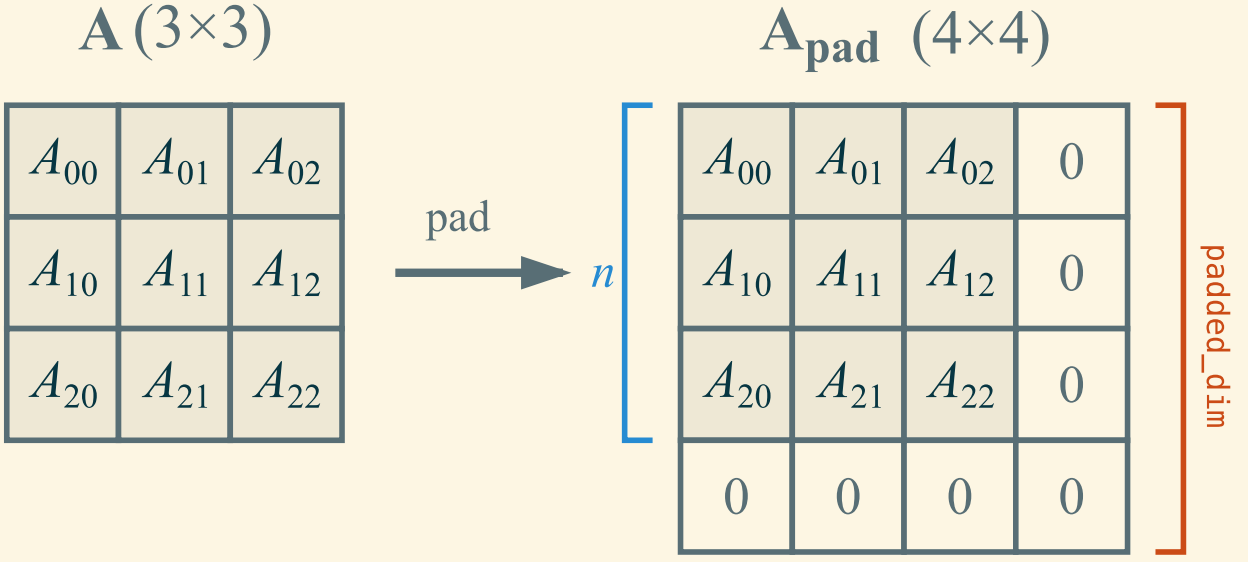
Problem: If n is not a power of 2 (e.g., $n = 3$), we cannot split evenly all the way down to 1×1 submatrices.

As the following slide explains, we solve this issue by **padding the input matrices with zeros** if necessary.

Home

DIVIDE AND CONQUER: PADDING

1. Find the smallest power of 2 that is $\geq n$. The C++ code calls this number `padded_dim`. For example, if $n = 3$, then `padded_dim = 4`.
2. Embed $\mathbf{A}$ and $\mathbf{B}$ in `padded_dim`-by-`padded_dim` matrices, filling extra elements with zeros.
3. Multiply the padded matrices using the recursive algorithm.
4. Extract the top-left $n \times n$ block of the result.



Home

WHY PADDING PRESERVES CORRECTNESS

Consider the product $C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$.

When we pad $\mathbf{A}$ and $\mathbf{B}$ with zeros to size `padded_dim` $> n$:

- **For $i, j < n$:**
  The extra terms in the sum (where $k \geq n$) are $A_{ik} \cdot B_{kj} = 0 \cdot B_{kj} = 0$. The result is unchanged.
- **For $i \geq n$ or $j \geq n$:**
  These elements of $\mathbf{C}$ are zeros. We discard them.

Conclusion: The top-left $n \times n$ block of the padded product equals the true product $\mathbf{A} \cdot \mathbf{B}$ of the unpadded matrices.

Home

WHAT YOU NEED TO IMPLEMENT

If you haven't already, download the starter files.

The starter code provides `matmulDivideConquer8()`, a wrapper function with padding logic already implemented.

Your task: Implement the recursive helper function `blockMatmulAdd()` that:

1. Handles the **base case** (1×1 submatrix).
2. Computes the **8 recursive calls** for the four quadrants of $\mathbf{C}$:
   - $\mathbf{C}_{11} \mathrel{+}= \mathbf{A}_{11}\mathbf{B}_{11}$, then $\mathbf{C}_{11} \mathrel{+}= \mathbf{A}_{12}\mathbf{B}_{21}$
   - Similarly for $\mathbf{C}_{12}, \mathbf{C}_{21}, \mathbf{C}_{22}$

Home

EXERCISE 2B: 8-FOLD BLOCK-RECURSIVE MATRIX MULTIPLICATION

To implement matrix multiplication recursively, complete the recursive helper function `blockMatmulAdd()` in `matmul_divide_conquer.cpp`, provided in the ZIP archive. The wrapper `matmulDivideConquer8()`, including padding and input checks, is already implemented.

Implementation trick: Do not copy $A_{11}, A_{12}, \ldots$ into new matrices. Instead, treat each submatrix as a view into the entire $n \times n$ matrix: represent it by `(row_offset, col_offset, size)` and access elements via index offsets.

To compile and test locally, run the following commands:

```
make clean
make check_divide_conquer
make run TARGET=check_divide_conquer FILE=test_0.txt
```

Upload your solution to Gradescope. Detailed instructions are provided on xSITe (Dropbox → Week 02 Lab Exercise 2b: Gradescope—8-Fold Block-Recursive Matrix Multiplication).

BENCHMARKING

Home

FROM THEORY TO MEASUREMENT

Asymptotic notation tells us how an algorithm's running time grows as the input size n increases. However, wall-clock time also matters: two $\Theta(n^3)$ algorithms can differ by a large constant factor, which can significantly affect real-world performance.

In the following exercise, you will **measure execution time** on your computer. You do not need to write timing code yourself: a provided program runs the algorithms, measures the time, and produces a table and a plot.

Your task is to interpret the results:

- Look for the **overall trend** as n increases.
- Notice that constant factors and overhead can matter in practice.

Small fluctuations from run to run are normal; focus on the pattern.

Home

EXERCISE 2C: BENCHMARKING AND ANALYSIS

Compile and run the provided benchmarking program to measure the running times of both matrix multiplication algorithms for various matrix sizes n:

```
make clean
make benchmark
./benchmark [max_n]
```

The optional argument `max_n` specifies the largest matrix dimension to benchmark (must be a power of 2, at least 8). The default is 64. For example, `./benchmark 256` tests sizes 8, 16, 32, 64, 128, and 256. The program will output a table of running times and generate a plot.

Your task: Submit a short report (PDF or DOCX) that includes:

1. The generated plot showing running time versus n for both algorithms.
2. The **theoretically expected running times** for each algorithm, with justification.
3. A discussion of whether the measurements **support the theoretical predictions**.
4. Upload your report to xSITe (Dropbox $\rightarrow$ Week 02 Lab Exercise 2c: Benchmarking and Analysis).

Home

CONCLUSION

Home

SUMMARY OF KEY LEARNING OUTCOMES

1. Used $O/\Omega/\Theta$ notation and the Master Theorem to classify the growth rates of divide-and-conquer recurrences. ↪
2. Built a working block-recursive matrix multiplication by expressing subproblems via offsets, decomposing into quadrants, and handling the base case correctly. ↪
3. Interpreted empirical performance differences between algorithms with the same $\Theta(n^3)$ scaling, attributing gaps to constant factors (e.g., call overhead and compiler behavior). ↪
4. Connected theory to data by benchmarking, checking scaling trends (e.g., when doubling n), and assessing whether the measurements align with the predicted asymptotics. ↪

Home

OUTLOOK

Challenges to explore on your own:

- Change the loop from i-j-k to i-k-j. Faster or slower? Why?
- Replace 8-fold recursion with Strassen's 7-fold recursion for $\Theta(n^{2.81})$. See Section 4.2 of Cormen *et al.* (2022).

Next week's topics:

This lab explored **divide-and-conquer** strategies and **asymptotic analysis** through matrix multiplication. Next week continues these themes with two classic sorting algorithms.

- **Quicksort:** A divide-and-conquer sorting algorithm—more practice analysing recurrences.
- **Heapsort:** Introduces **heaps**, a data structure that efficiently maintains a collection when elements are inserted or removed.

BIBLIOGRAPHY

- Cormen, T.H. *et al*. (2022) *Introduction to algorithms*. 4th ed. MIT Press.