Data Structures

Week 3 – Arrays: one- and multidimensional arrays, storage, dynamic arrays

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Big-O notation:** the order of the number of steps – O(1), O(log n), O(n), O(n^2)
- **Linear search:** O(n), on any array
- **Binary search:** O(log n), on a sorted array, **direct access** needed

Questions for today

- How does the machine store an array, and why is access O(1)?
- How much does inserting or deleting an element cost?
- How can a static array be “extended”?

The concept of an array

An **array** is a finite sequence of elements of the same type, accessed by an **index**.

Property	Array
Element type	**homogeneous**
Size	**static** – fixed at creation
Storage	**contiguous** – the elements follow each other directly
Access	**direct** – any element in O(1) time
Relationship	**linear**

Also called: **vector** (one-dimensional), **matrix** (two-dimensional)

Arrays in Java and Go

Java

```java
int[] a = new int[5];          // filled with 0
int[] p = {2, 3, 5, 7, 11};
p[0] = 1;
int n = p.length;              // 5

for (int x : p)
    System.out.print(x + " ");

int[] b = a;                   // the SAME array
int[] c = Arrays.copyOf(p, p.length);
```

- The array is an **object**, the variable is a reference
- Default values: `0`, `0.0`, `false`, `null`

Go

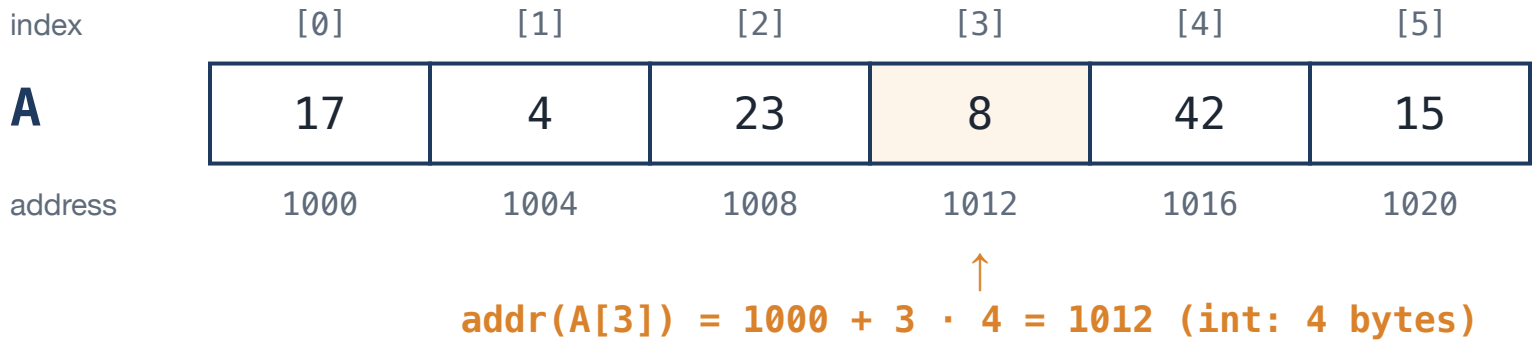
```go
var a [5]int                   // filled with 0
p := [5]int{2, 3, 5, 7, 11}
p[0] = 1
n := len(p)                    // 5

for _, x := range p {
    fmt.Print(x, " ")
}

b := a                         // a COPY!
s := []int{2, 3, 5}            // slice
```

- The size is **part of the type**: `[5]int` ≠ `[6]int`
- An array is a **value** – in practice we use slices (`[]int`)

Storage in memory



The address of the **i**-th element is obtained by **a single calculation**:

> addr(A[i]) = base address + i · element size (indexing from 0)
> addr(A[i]) = base address + (i – 1) · element size (indexing from 1)

That is why access is **O(1)** – there is no need to walk through the previous elements.

Why does indexing start at 0?

- With 0-based indexing the index is the element's **offset** from the start of the array
- `A[0]` is at the base address, `A[3]` is three elements further
- C, Java, Go, Python: from 0 · Pascal, Fortran, Lua, mathematics: from 1

> In pseudocode we index **from 1**, in programs **from 0**. When translating:
> `For i ← 1 to n` → `for (int i = 0; i < n; i++)`

Bounds checking

- Java: `ArrayIndexOutOfBoundsException` · Go: `panic: index out of range`
- C has no check: the program silently **overwrites** neighbouring memory – a source of serious security bugs

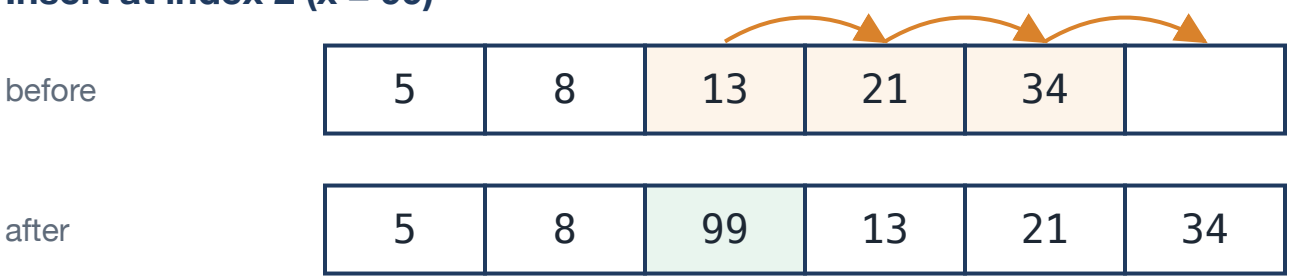
Cost of the operations

Operation	Steps	Why?
reading / writing the **i**-th element	**O(1)**	address calculation
search (unsorted)	O(n)	linear search
search (sorted)	O(log n)	binary search
insertion at the end (if there is room)	O(1)	
insertion at the beginning / in the middle	**O(n)**	elements must be shifted
deletion at the beginning / in the middle	**O(n)**	elements must be shifted

> An array is **fast for reading** and **slow for modification** – if we insert and delete a lot in the middle, a **linked list** (week 9) may be a better choice.

Insertion and deletion

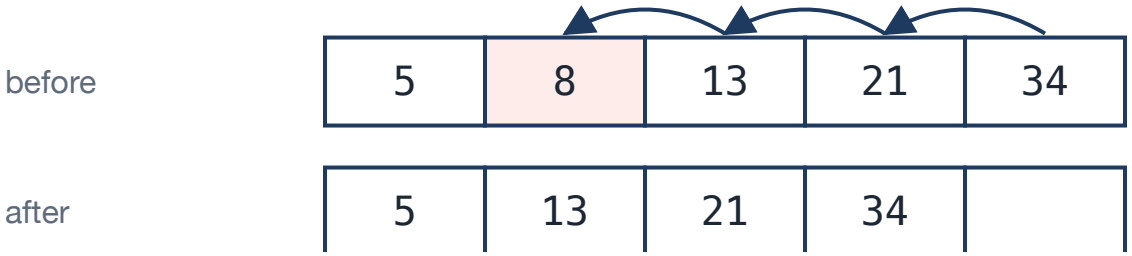
Insert at index 2 (x = 99)



3 elements shifted right

(starting from the back!)

Delete at index 1



3 elements shifted left

(starting from the front)

Insertion and deletion – algorithm

Insertion at position k

```
Procedure Insert(A, n, k, x)
  // precondition: n < capacity
  For i ← n down to k
    A[i + 1] ← A[i]
  End for
  A[k] ← x
  n ← n + 1
End procedure
```

Deletion at position k

```
Procedure Delete(A, n, k)
  For i ← k to n − 1
    A[i] ← A[i + 1]
  End for
  n ← n − 1
End procedure
```

Java

```java
static int insert(int[] a, int n,
                  int pos, int x) {
    for (int i = n - 1; i >= pos; i--)
        a[i + 1] = a[i];
    a[pos] = x;
    return n + 1;
}
```

When inserting, shifting must start **from the back**! Going from the front, every element would be overwritten by the first shifted value.

Reversing and rotating in place

Reversing with two indexes – O(n), O(1) space

```
Procedure Reverse(A, n)
  i ← 1 ; j ← n
  While i < j
    swap(A[i], A[j])
    i ← i + 1 ; j ← j − 1
  End while
End procedure
```

`[1 2 3 4 5]` → `[5 4 3 2 1]`

Rotating left by k positions – with three reversals

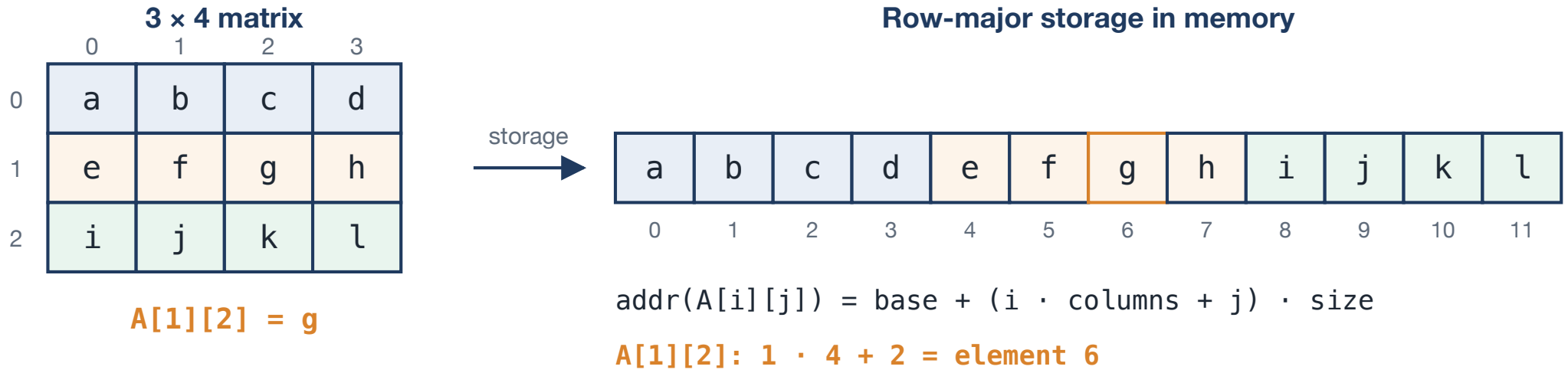
`[1 2 | 3 4 5]`, k = 2

1. reverse the first k elements: `[2 1 | 3 4 5]`
2. reverse the rest: `[2 1 | 5 4 3]`
3. reverse the whole array: `[3 4 5 1 2]` ✓

Without a helper array, in O(n) steps.

Multidimensional arrays

Two-dimensional array (matrix)



- Memory is **one-dimensional** – the matrix must be “flattened” into a sequence
- **Row-major** storage (C, Java, Go): first the first row, then the second …
- **Column-major** storage (Fortran, MATLAB): first the first column …

Address calculation: the general formula

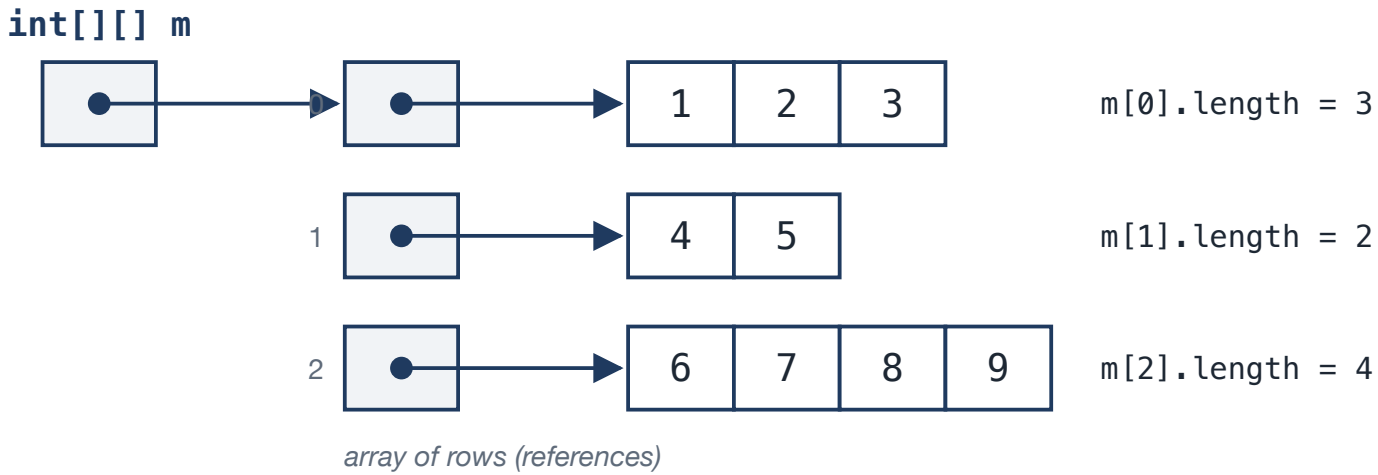
An **N** × **M** matrix (N rows, M columns), element size **s**, base address **k**:

	indexing from 0	indexing from 1
Row-major	$k + (i \cdot M + j) \cdot s$	$k + ((i - 1) \cdot M + (j - 1)) \cdot s$
Column-major	$k + (j \cdot N + i) \cdot s$	$k + ((j - 1) \cdot N + (i - 1)) \cdot s$

> **Example:** `int A[10][20]` (4-byte elements), base address 2000, row-major, indexing from 0.
> addr(A[3][5]) = 2000 + (3 · 20 + 5) · 4 = 2000 + 260 = **2260**

Three-dimensional array `A[i][j][l]`, dimensions N × M × L: $k + ((i \cdot M + j) \cdot L + l) \cdot s$

2D arrays in Java: an array of arrays




```
int[][] m = new int[3][4];          // 3 rows, each of length 4
int[][] jagged = { {1, 2, 3}, {4, 5}, {6, 7, 8, 9} };
```

- Every row is a **separate object** → the rows may have different lengths (**jagged array**)
- `m.length` is the number of rows, `m[i].length` is the length of the **i**-th row

2D arrays in Go

True 2D array – one contiguous block

```
var grid [3][4]int
grid[1][2] = 7
```

- The size is known at compile time
- Exactly row-major storage

Slice of slices – like in Java

```
m := make([][]int, rows)
for i := range m {
    m[i] = make([]int, cols)
}
```

- The size can be given at run time
- The rows may have different lengths

When traversing a matrix, the **row-major** order (outer loop over the rows) is faster, because it makes better use of the processor cache.

Matrix operations

Main and anti-diagonal – O(n)

```
For i ← 1 to n
  main ← main + A[i, i]
  anti ← anti + A[i, n − i + 1]
End for
```

Transposition – O(n · m)

```
For i ← 1 to n
  For j ← 1 to m
    T[j, i] ← A[i, j]
  End for
End for
```

Matrix multiplication – (n×k) · (k×m), **O(n · k · m)**

```
For i ← 1 to n
  For j ← 1 to m
    C[i, j] ← 0
    For x ← 1 to k
      C[i, j] ← C[i, j] + A[i, x] · B[x, j]
    End for
  End for
End for
```

For square matrices **O(n³)** – three nested loops.

Dynamic arrays

The problem: the size of an array is fixed

- We often **do not know in advance** how many elements must be stored (e.g. data read from input)
- Too small array → it does not fit · too large array → waste

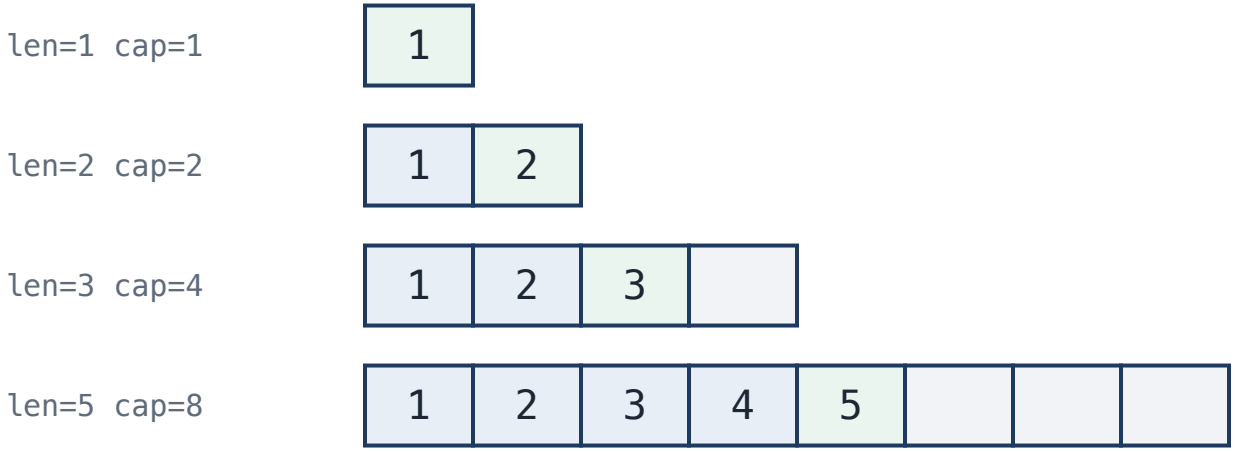
Solution: dynamic array – an array that is **replaced by a larger one** when necessary

- `size` (length): how many elements it actually contains
- `capacity` : the size of the backing array

Java: `ArrayList` · Go: slice + `append` · C++: `vector` · Python: `list`

How a dynamic array works

append: when full, allocate an array twice as large and copy the elements



copies: 1 + 2 + 4 + … < 2n

→ **one append is O(1) on average**

(amortized cost)

- Growing by one each time: 1 + 2 + 3 + … + n = **O(n²)** copies in total
- **By doubling:** 1 + 2 + 4 + … + n < 2n → O(n) in total, **O(1)** per element

Our own dynamic array

Java

```java
class DynamicArray {
    private int[] data = new int[1];
    private int size = 0;

    void add(int x) {
        if (size == data.length) {
            int[] bigger =
                new int[2 * data.length];
            for (int i = 0; i < size; i++)
                bigger[i] = data[i];
            data = bigger;
        }
        data[size++] = x;
    }
    int get(int i) { return data[i]; }
}
```

Go

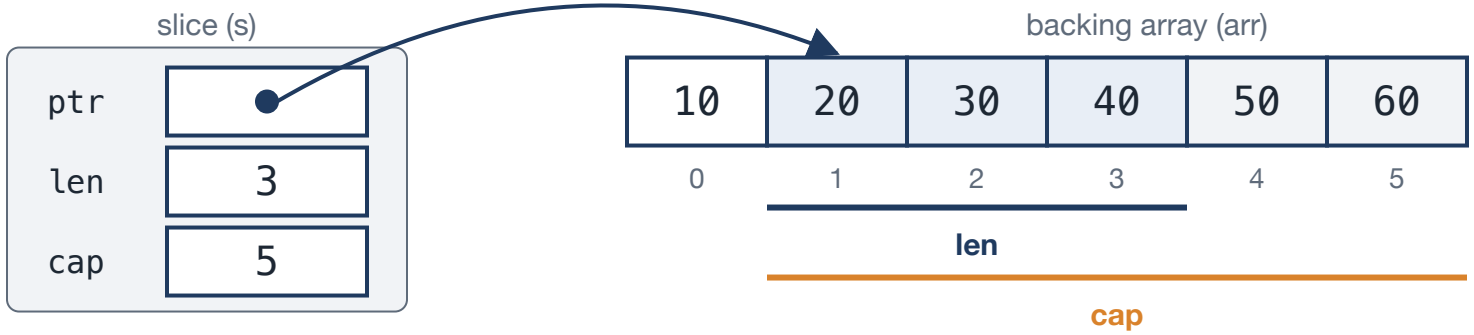
```go
type DynamicArray struct {
    data []int
    size int
}

func (d *DynamicArray) Add(x int) {
    if d.size == len(d.data) {
        bigger := make([]int, 2*len(d.data))
        copy(bigger, d.data[:d.size])
        d.data = bigger
    }
    d.data[d.size] = x
    d.size++
}
```

Measurement: 1 000 000 `add` calls → **1 048 575** copies, **1.05** per element

Go slice: length, capacity, backing array

```
s := arr[1:4] // len 3, cap 5
```




```
s[0] = 99 → arr[1] becomes 99 too! (shared backing array)
```

- A slice consists of three values: a **pointer** to the backing array, the **length** (`len`) and the **capacity** (`cap`)
- `append` allocates a new array only when the capacity runs out
- That is why we always write: `s = append(s, x)` – the result must be assigned back!

Java `ArrayList`

```
ArrayList<Integer> list = new ArrayList<>();
list.add(5);          // at the end — amortized O(1)
list.add(0, 3);       // insertion at the beginning — O(n)
list.get(1);          // reading — O(1)
list.set(1, 9);       // writing — O(1)
list.remove(0);       // deletion by index — O(n)
list.size();          // number of elements
```

- Stores **objects** only: `Integer` instead of `int` (automatic boxing and unboxing)
- Internally an `Object[]` array, grown by a factor of 1.5 when full

Summary

- The array is a **homogeneous, static, contiguous, direct-access** data structure
- Address calculation: `base address + index · element size` → access **O(1)**
- Insertion and deletion in the middle are **O(n)** – elements must be shifted
- Matrix: **row-major** storage, address = k + (i · M + j) · s
- Java: a 2D array is an array of arrays · Go: `[N][M]T` or `[][]T`
- **Dynamic array:** doubling the capacity → appending is amortized **O(1)**

Next week: strings – sequences of characters, encoding, operations, pattern matching.

Review questions

1. True or false: the elements of an array are scattered in memory.
2. How many steps does it take to delete the first element of an **n**-element array?
3. An `int A[5][8]` matrix is stored row-major from address 100, with 4-byte elements, indexing from 0. What is the address of `A[2][3]`?
4. Why must shifting start from the back when inserting?
5. Why do we double the capacity of a dynamic array instead of increasing it by one?
6. What does it print? `s := []int{1,2,3}; t := s; t[0] = 9; fmt.Println(s[0])`

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Basics	`ArrayBasics.java`	`w03/arraybasics`	creation, default values, bounds, copying
Operations	`ArrayOps.java`	`w03/arrayops`	insertion, deletion, reversal, rotation
Matrix	`Matrix.java`	`w03/matrix`	diagonals, transposition, multiplication, jagged array
Dynamic array	`DynamicArray.java`	`w03/dynarray`	doubling the capacity, number of copies, `append`

In the `dynarray` Go program, observe how the capacity of the built-in `append` grows, and what happens with a shared backing array!

✎ Exercises

1. **Average:** compute the average of the elements of an integer array!
2. **Contains:** decide whether the array contains a given value!
3. **Second largest:** find the second largest (distinct) element with a **single** pass!
4. **Removing duplicates:** remove the duplicates from a **sorted** array in place, and return the new length! (`[1,1,2,3,3,3,5] → [1,2,3,5]`)
5. **Merging:** merge two sorted arrays into one sorted array in O(n + m) steps!
6. **Row and column sums:** compute the sum of every row and column of a matrix!
7. **Spiral:** print the elements of a matrix in spiral order! (`1 2 3 6 9 8 7 4 5`)

Solutions: `Exercises03.java`, `w03/exercises`

✎ Homework

1. Extend our own `DynamicArray` class with `insert(i, x)` and `remove(i)` operations! How many steps do they take?
2. Write `remove` so that the capacity is halved when the array becomes **a quarter** full! Why not halve it when it is half full?
3. Write a program that rotates an **n** × **n** matrix **in place** by 90 degrees clockwise! (Hint: transposition + reversing the rows.)
4. Compute: `double T[50][40]` (8-byte elements) column-major, indexing from 1, base address 5000. What is the address of `T[12][7]`?