

Matrix operations: multiplication

Task: Multiply square matrices A and B of size N1.

One output entry uses a row of A and a column of B.

$$C[i][j] = \sum_{k=0}^{N1-1} A[i][k] B[k][j]$$

Keep fixed	Vary
Row i of A	Column k of A
Column j of B	Row k of B

Reset sum to zero for each $C[i][j]$, then accumulate products.

Multiplication rule from `matrixmult-run.c`, specialized to the square-matrix exercise in `matrixmult.c`; replaces the missing figure.

Matrix multiplication: complete the inner loop

```
for (i = 0; i<N1; i++) {  
    for (j = 0; j<N1; j++) {  
        sum = 0;  
        for (k=0; k<N1; k++) {  
            /* fill in your code here */  
        }  
        C[i][j] = sum;  
    }  
}
```

Assume a positive, initialized N1.

A and B are initialized square matrices; C stores the result.

What statement belongs in the innermost loop?

Loop excerpt from `matrixmult.c`; the exercise blank is retained. Its uninitialized `const int N1`; declaration is omitted: set the dimension before declaring the arrays. Assume `i`, `j`, `k`, `sum` are integers.

Fill both blanks so that every element of $A = B + C$ is computed.

Assume B and C are initialized integer matrices, each with 2 rows and 3 columns.

```
int A[2][3];  
for (int i = 0; i < 2; i++)  
    for (int j = 0; j < /* BOUND */; j++)  
        A[i][j] = /* EXPR */;
```

Poll response: Type BOUND = ...; EXPR = ...

Give a number and an expression in i and j.

1 mark

Let $C = AB$, using matrix multiplication. What is $C[1][0]$?

```
int A[2][2] = {{1, 2}, {3, 4}};
```

```
int B[2][2] = {{2, 1}, {1, 3}};
```

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$$

Poll response: Submit one integer: the value of $C[1][0]$.

Both indices start at 0.

1 mark

This code tries to compute row 0 of $C = AB$. What is actually printed?

```
int A[2][2] = {{1, 2}, {3, 4}};
int B[2][2] = {{2, 1}, {1, 3}};
int C[2][2], sum = 0;
for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 2; k++)
        sum += A[0][k] * B[k][j];
    C[0][j] = sum;
}
printf("%d", C[0][1]);
```

Poll response: Submit the single integer printed by this code.

Trace the code as written, without fixing it.

1 mark

Mini-quiz 4 | Answers and reasoning

M4: `BOUND = 3; EXPR = B[i][j] + C[i][j].`

Visit all three columns and add corresponding entries.

M5: 10 Use row 1 of A and column 0 of B:

$$C[1][0] = 3 \times 2 + 4 \times 1 = 10.$$

M6: 11 After column 0, sum is 4. The next column adds $1 \times 1 + 2 \times 3 = 7$, giving 11 instead of 7.

Fix: reset `sum = 0`; inside the `j`-loop, immediately before the `k`-loop.

Marking

M4: half a mark per blank; accept equivalent expressions. M5 and M6: one mark each. Total: 3 marks.

This week : Take Aways

- **Technical:** Learned about single, multidimensional arrays, character arrays, strings. Declaration, Initialization, reading, writing.
- **Problem Solving :** Writing programs to solve various tasks associated where use of arrays, matrices, strings are natural.

This week : Take Aways

- **Technical:** Learned about single, multidimensional arrays, character arrays, strings. Declaration, Initialization, reading, writing.
- **Problem Solving :** Writing programs to solve various tasks associated where use of arrays, matrices, strings are natural.
- **Meta-level message about the approach :** Writing algorithms/pseudo-code/programs - identify simpler tasks within the given task, solve them and then try to combine them to get the bigger solution. (Eg : using the code for "finding largest in a subarray" to do "sorting")

This week : Take Aways

- **Technical:** Learned about single, multidimensional arrays, character arrays, strings. Declaration, Initialization, reading, writing.
- **Problem Solving :** Writing programs to solve various tasks associated where use of arrays, matrices, strings are natural.
- **Meta-level message about the approach :** Writing algorithms/pseudo-code/programs - identify simpler tasks within the given task, solve them and then try to combine them to get the bigger solution. (Eg : using the code for "finding largest in a subarray" to do "sorting")
- **Observation:** Subtasks that appear once solved, can be used in several parts of the program.

This week : Take Aways

- **Technical:** Learned about single, multidimensional arrays, character arrays, strings. Declaration, Initialization, reading, writing.
- **Problem Solving :** Writing programs to solve various tasks associated where use of arrays, matrices, strings are natural.
- **Meta-level message about the approach :** Writing algorithms/pseudo-code/programs - identify simpler tasks within the given task, solve them and then try to combine them to get the bigger solution. (Eg : using the code for "finding largest in a subarray" to do "sorting")
- **Observation:** Subtasks that appear once solved, can be used in several parts of the program.
Functions !!

Weeks 1–7

- C data types, operators, input and output.
- Conditionals and loops.
- Numerical methods, arrays and strings.

Where we are: Week 8

Weeks 1–7

- C data types, operators, input and output.
- Conditionals and loops.
- Numerical methods, arrays and strings.

Week 8

- Insertion sort and an implementation with fewer assignments.
- Functions: declaration, definition and calls.
- Blocks, scope, local and global variables.

Insertion sort

Before pass i , positions 0 through $i-1$ are already sorted. Insert $A[i]$ into this sorted portion.

Why start at $i = 1$?

The first element alone is already sorted.

Move $A[i]$ left by swapping with each larger neighbour.

After the pass, positions 0 through i are sorted.

```
for (int i = 1; i < n; i++) {  
    int j = i;  
    while (j > 0 &&  
           A[j-1] > A[j]) {  
        int temp = A[j];  
        A[j] = A[j-1];  
        A[j-1] = temp;  
        j--;  
    }  
}
```

Sorting order: **non-decreasing** (smallest first).

Insertion sort: tracing the swaps

At pass $i = 3$, the prefix 3 12 15 is sorted. Insert 8.

Action	Array contents
Start with $j = 3$.	3 12 15 8
$15 > 8$: swap $A[2]$ and $A[3]$. Decrement j to 2.	3 12 8 15
$12 > 8$: swap $A[1]$ and $A[2]$. Decrement j to 1.	3 8 12 15
$3 > 8$ is false. Stop.	3 8 12 15

The element 8 moves left twice. The sorted prefix now includes it.

Improved insertion sort

Save the element in `key`. Shift each larger element one position right. Finally, insert `key` into its correct position.

Example: insert 8

3 12 15 8

Save `key = 8`.

Start with `j = 2`.

The first shift overwrites the original location of 8. Saving `key` preserves its value.

```
for (int i = 1; i < n; i++) {  
    int key = A[i];  
    int j = i - 1;  
    while (j >= 0 &&  
           A[j] > key) {  
        A[j+1] = A[j];  
        j--;  
    }  
    A[j+1] = key;  
}
```

Improved insertion sort: tracing the shifts

Pass $i = 3$. Keep key = 8 unchanged throughout.

Action	Actual array contents
Save key = $A[3]$. Set $j = 2$.	3 12 15 8
$15 > 8$: $A[3] = A[2]$. Decrement j to 1.	3 12 15 15
$12 > 8$: $A[2] = A[1]$. Decrement j to 0.	3 12 12 15
$3 > 8$ is false. Stop with $j = 0$.	3 12 12 15
$A[j+1] = \text{key}$: write 8 at index 1.	3 8 12 15

The gap is conceptual. Old values remain until overwritten, so temporary duplicates in the array are expected.

Why insert at $j + 1$?

Case 1: $A[j] \leq \text{key}$

The element at j can stay before key . Every element shifted to its right is larger than key . Insert immediately after j .

For 3 12 15 8: $j = 0$, so insert 8 at index 1.

Case 2: $j == -1$

Every element in the sorted prefix was larger than key . The key belongs at the beginning.

For 3 12 15 1: all three elements shift right.

Then $j = -1$, so insert 1 at index 0.

Both cases use the same statement:

$A[j+1] = \text{key};$

The bounds check comes first

Version with swaps

```
j > 0 && A[j-1] > A[j]
```

At $j = 0$, there is no left neighbour. The first condition is false, so C skips the comparison that would read $A[-1]$.

Version with shifts

```
j >= 0 && A[j] > key
```

After shifting $A[0]$, decrementing j gives -1 . Again, the first condition prevents a read of $A[-1]$.

C evaluates `&&` left to right. When the first operand is false, it does not evaluate the second operand.

Improvement: fewer value assignments

Suppose the element being inserted must move left by k positions.

Version	Assignments that copy element values
Adjacent swaps	$3k$: each swap copies a value into temp, then updates two array positions.
Shifting	$k + 2$: save key once, perform k shifts, then insert key once.

In our example, $k = 2$:

Two swaps use $3 \times 2 = 6$ assignments.

Shifting uses $1 + 2 + 1 = 4$ assignments.

These counts exclude loop-index updates. Shifting uses fewer value assignments when $k > 1$; the counts are equal when $k = 1$.