

CW2 and Revision

Song Liu (song.liu@bristol.ac.uk)

GA 18, Fry Building,
Microsoft Teams (search "song liu").

Assessed Coursework 2

Deadline: Friday, Week 12.

Task: Given a data set ([MNIST](#)) containing images of handwritten digits, implement a simple classification algorithm ([k-nearest neighbor](#)), which labels a test image with "1", or "not 1"

Dataset:

0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9 9 9 9

Input: Input:

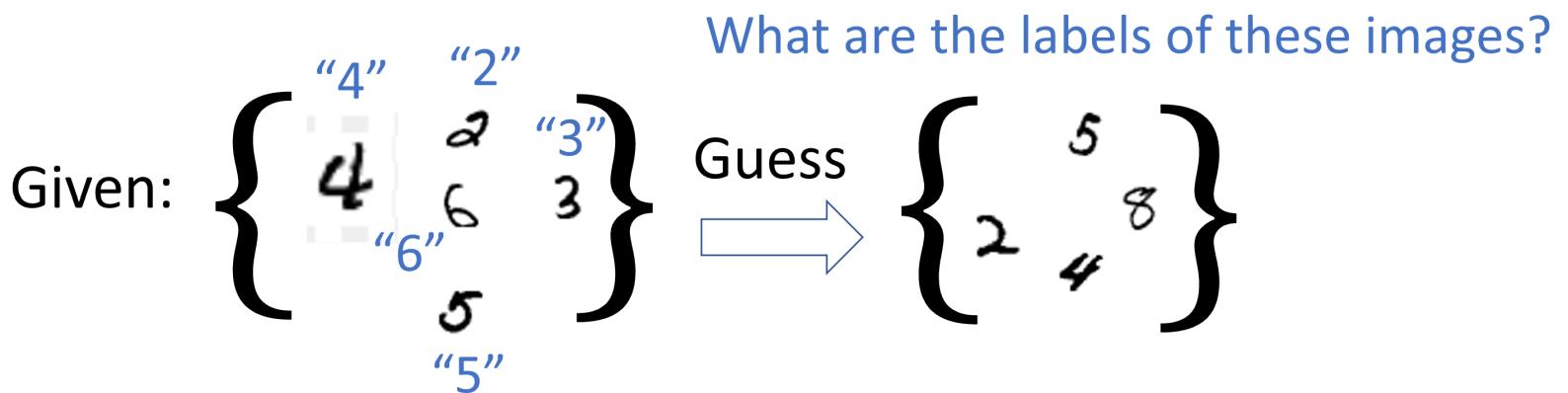


Output: Output:

"4" "1"

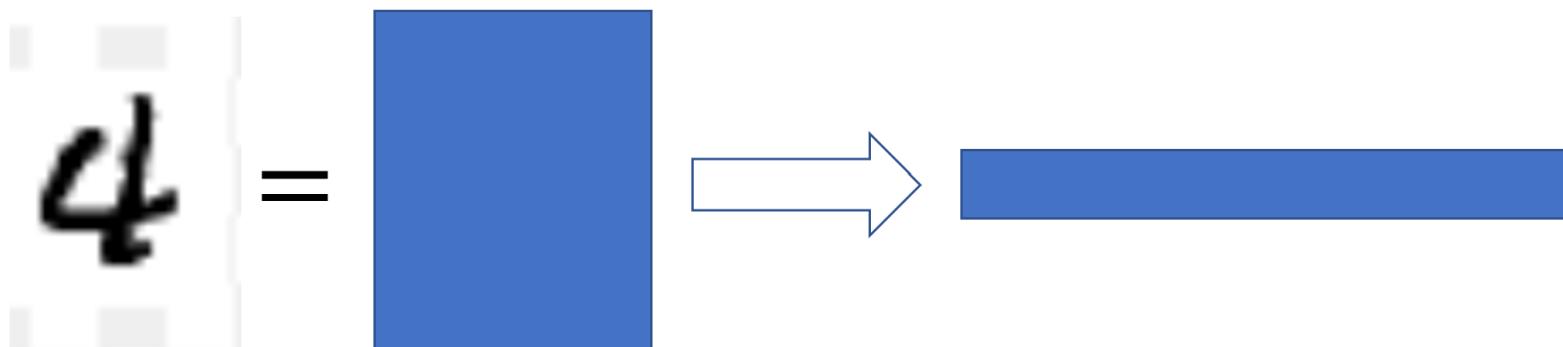
Assessed Coursework 2

- In this CW, the goal is to use images and labels in one dataset to predict labels of images in another dataset, where labels are NOT observed.



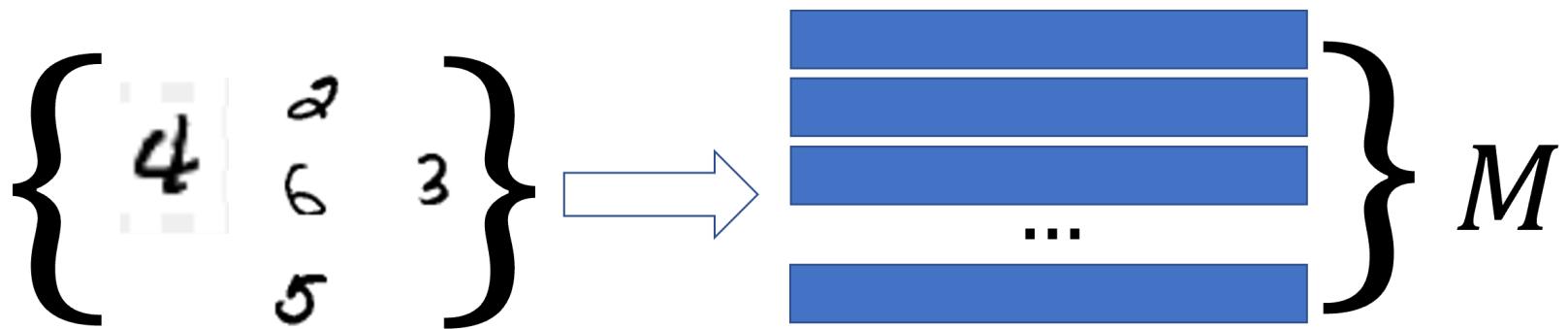
Assessed Coursework 2

- As we mentioned in previous lectures/labs, images are stored as flattened matrices (in row/col major order) in the memory.
- Each image is stored as a vector in this coursework.



Assessed Coursework 2

- In this CW, we are dealing with sets of images. Stacking all the image vectors together, you get a matrix.

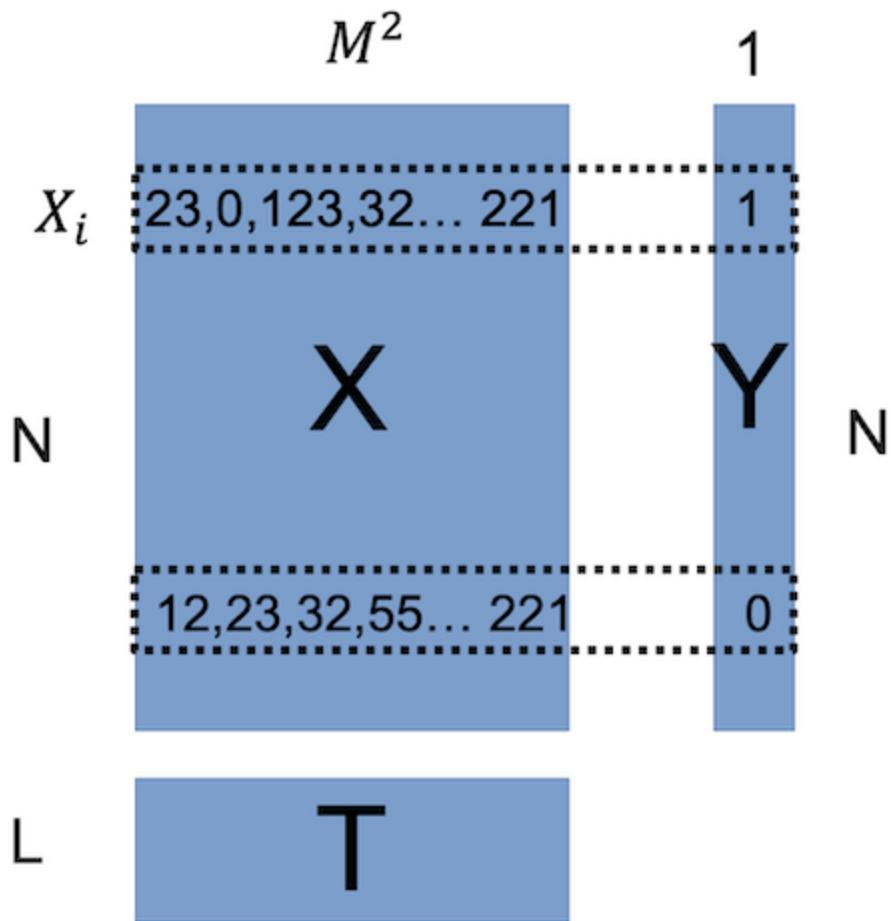


- Specifically, there are two sets of images in this CW, so they are represented by two matrices X, T .

Part I, The Data Set

- CW folder contains 3 `.matrix` files storing 3 matrices.
 - `X.matrix` stores an N by M^2 matrix X where each row is a grayscale M by M image stored in row-major order.
 - `T.matrix` stores an L by M^2 matrix T where each row is an M by M test image in row major order.
 - `Y.matrix` stores an N by 1 matrix Y where each row is a scalar, indicating whether the corresponding row in X is digit 1 or not.
 - X and Y together are called "training set" in machine learning, while T is the "testing set". Y is called the "labels" of X .

Part I, Data Structure



- If $Y_i = 1$, then the image X_i is a handwritten digit 1. If $Y_i \neq 1$, the image X_i is NOT a handwritten digit 1.

Part I, Loading Dataset

- The code for loading these images from files have been provided to you. Matrices are represented by a **matrix structure** in this coursework.

```
struct matrix
{
    int numrow; //number of rows
    int numcol; //number of columns
    int *elements; // pointer pointing to an integer array
    // storing all entries in the matrix in row major order.
};

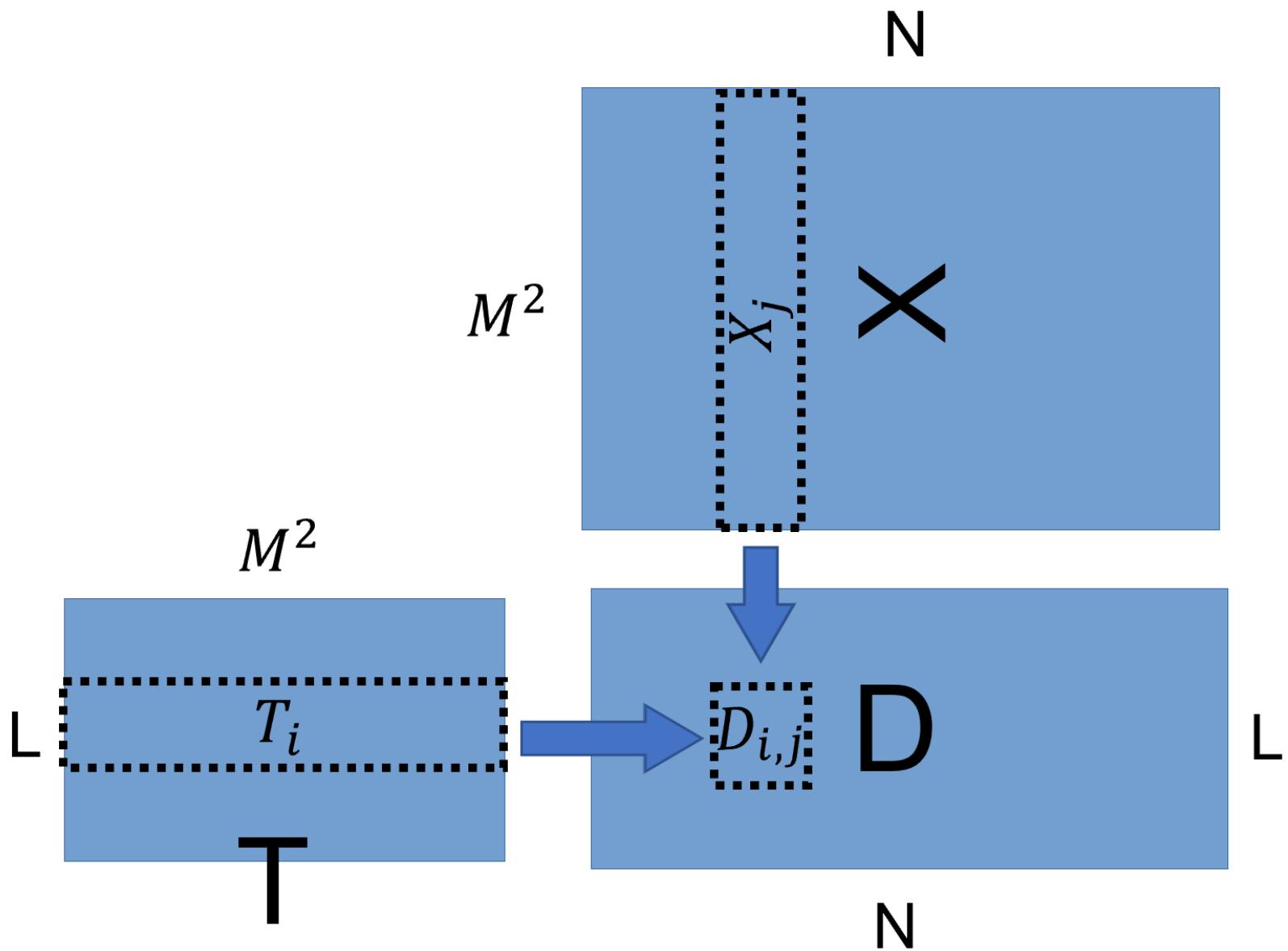
typedef struct matrix Matrix;
// now "Matrix" is an alias of "struct matrix"
```

- By simply running the skeleton code, you should see some basic statistics of X , Y and T .
 - What are M , N and L ?
 - How many images in the training set X are digit 1?

Part II Computing Distance Matrix D (40pt in total)

- Construct an L by N matrix D , where the i, j -th element $D_{ij} = \text{dist2}(T_i, X_j)$
 - T_i is the i -th row of T .
 - X_j is the j -th row of X
- $\text{dist2}(a, b)$ computes the squared euclidean distance between two vectors a and b with K elements.
 - $\text{dist2}(a, b) := \sum_{k=1}^K (a_k - b_k)^2$.

Part II (Computing D)



Part II.1 (15pt) Constructing D

Before your `main` function,

1. Write a few helper functions:

- `int get_elem(Matrix M, int i, int j)`

returns the `i, j` th element of matrix `M`.

- `void set_elem(Matrix M, int i, int j, int value)`

assign `value` to the `i, j` th element of matrix `M`

In this coursework, `i, j` are zero-based indices.

In your `main` function,

2. Allocate HEAP memory for D .

3. Declare and initialize a new `matrix` variable `D`.

Part II.2 Computing D (25pt)

Now, populate the matrix D with correct values.

- Hint: Compare the computation of D and the matrix multiplication. What are the similarities and what are the dissimilarities?
 - Can you modify the matrix multiplication code to compute matrix D ?
- Hint, you can write a function

```
void pairwise_dist(Matrix T, Matrix X, Matrix D)
```

- where D is the output, storing the outcome.
- Partial points will be given for correctly written code for computing $\text{dist2}(a, b)$.

Part III.1 Guessing Labels (15%)

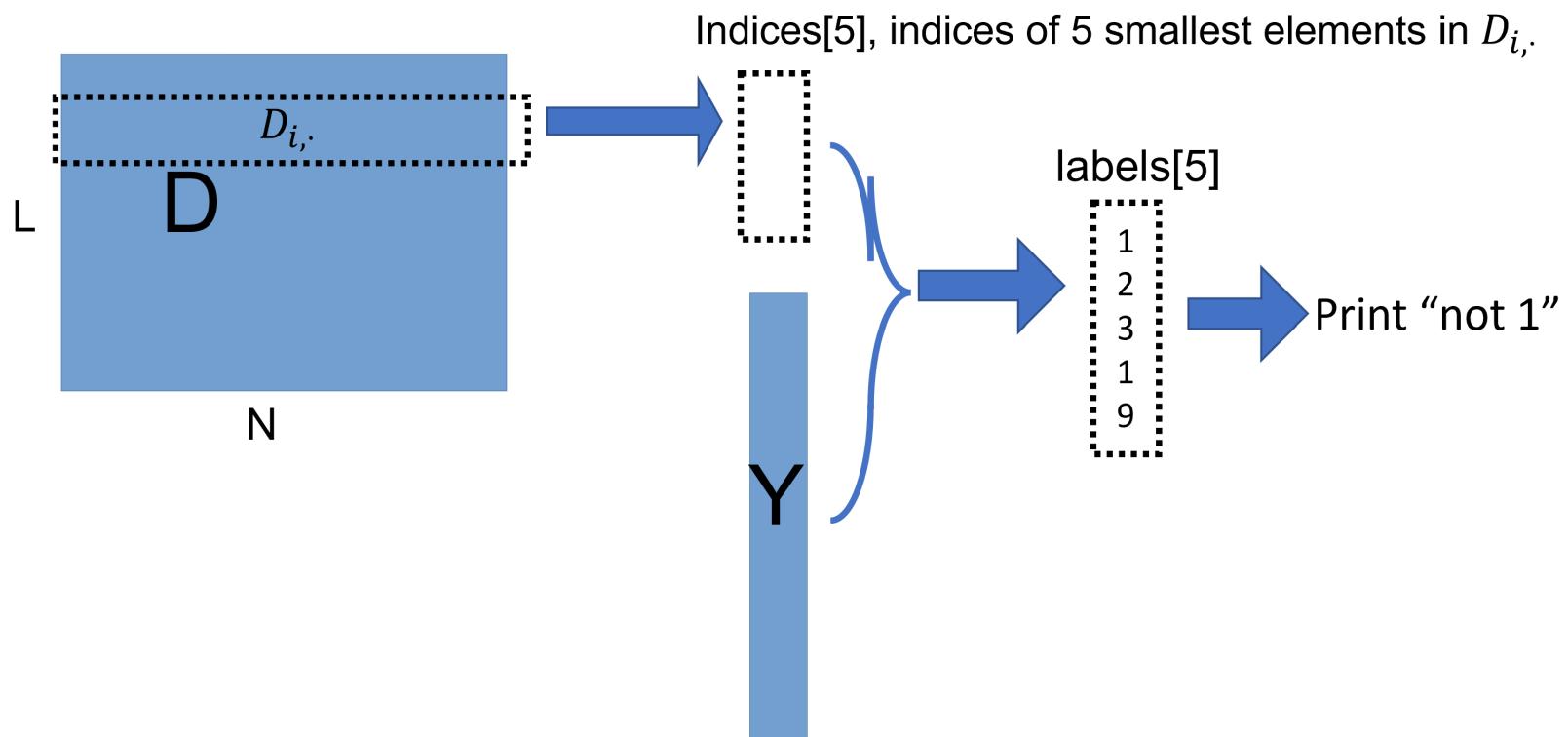
- For each row of matrix D , find the indices of the five smallest elements.
 - Suppose the i -th row of D is
 $[3, 2, 5, 1, 2, 5, 13, 46, 32],$
 - The indices of the five smallest elements are
 $[3, 1, 0, 5, 6].$

Part III.2 Guessing Labels (15%)

- Now, suppose the array `indices` contains the indices of the five smallest elements in row i of matrix D .
 - Create a new array `labels` with length 5.
 - Assign the value of $Y_{\text{indices}[k],0}$ to the k -th element in `labels`.
 - Count the number of `1` in `labels`.
 - If the count $>= 3$, print out `1`. Otherwise, print `not 1`.
- Repeat above for all rows in D .

Part III Guessing Labels (30pts total)

For each row in D , do:



Part III Guessing Labels

- At each row D_i , the print-out is your "guess" of the testing image T_i using 5-nearest neighbour algorithm.
 - If the print-out is `1`, it means the algorithm thinks the image T_i is a digit 1.
 - If the print-out is `not 1`, it means the algorithm thinks the image T_i is NOT a digit 1.
- After your guess, you can optionally print the image stored in T_i to the console to validate your guess.

Part III Guessing Labels

- Hint: Write a helper function

```
void minimum5(int len, int a[], int indices[])
```

It takes an array `a` with length `len` as input, then fills `indices[]` with the indices of the five smallest elements.

- You might want to test your functions in a separate c file to ensure that they are correctly written.

Final Project: Marking Criteria

- Submitting correct code (10%)
 - Submitting a C file with **the correct name**.
 - Your code compiles and runs **without major error** such as **crash, infinite loop**.
 - It will be tested using `gcc` in the lab pack.
- Part II 40% (15% + 25%)
- Part III 30% (15% + 15%)
- Good Coding Practice (20%)
 - Good code format
 - Good variable naming scheme.
 - Apt comments

Final Project: Dos and Don'ts

- You can discuss with your classmates about general strategies but write your own code!
- Don't give your code to other students.
- Review relevant previous lab sessions before you start.
- You can use whatever material you can find to help you complete the task, but you need to add a reference in the comments.
- You are only allowed to use standard features of C.
 - You can use `stdio.h`, `stdlib.h`, `limits.h` and `math.h`.
 - If you want to use other libraries, consult with the lecturer or TA beforehand.

Final Project: Q&A

- We will answer questions posted on the Blackboard forum or answering them during the lab sessions.
- We will inspect the forum regularly and try to respond in 24 hours.

Revision

Look back:

- TB1: C Programming Language (done!)
- TB2: 50/50
 - Some aspects of C++ Programming Language.
 - R Programming Language.

Revision

- Basics of Computing:
 - Foundation of Computing
 - Functions
 - Flow Control
 - Recursion and Stack Memory
- Advanced Language Features
 - Arrays
 - Pointers
 - Dynamic Memory Allocation
 - Structure and IO

Foundation

von Neumann Architecture

- Central Processing Unit (CPU)
 - Performs computational tasks.
 - Controls Input/Output (IO) devices.
 - Maintains data stored in the memory.
- Memory
 - Stores program/data being used by CPU temporarily.
- IO Devices
 - Hard disk
 - Display
 - Camera
 - Touch Screen, etc.

Low-level Programming Language

- **Advantages of Low-level Programming Language:**
 - gives coder total control of hardware.
 - can be efficient since it needs no translation (You talk to the computer using its native language!).
- **Disadvantages of Low-level Programming Language:**
 - can damage the hardware if the coder is not careful.
 - is difficult to learn and read (machine instructions are usually very different from human languages).
 - only works on a specific cpu architecture (e.g. x86 or ARM). Thus the code is not "portable".

High-level Programming Language

- Advantages of High-level Programming Languages
 - Close to human language, easy to learn/read.
 - CPU architecture independent, a.k.a., "Portable".
- Disadvantages of High-level Programming Languages
 - Less efficient as the code requires "translation" before it can be executed on CPU.
 - Cannot directly communicate with hardware: the coder has to interface with *the abstraction*.

Functions

Functions in Programming

- Functions are individual building blocks of your program that accomplish specific tasks.
 - Function helps you divide your code into smaller, more **manageable and readable** pieces.
 - Code in a function is only executed when its host function is "called".
 - In our "Hey Jude" example, `main` is the caller of `verse1` and `verse2`
- Some functions take *input arguments* from the caller.
- Some functions return *an output value* to the caller after all its code are executed.
- Some functions do not have input or output.

Your Own Function

You can write your own function and call it from `main()`:

```
...
void sayhello(){
    printf("Hello World!\n");
}
int main(){
    sayhello(); //calling "sayhello" function.
    return 0;
}
```

Data Types in C

- Some data types are:
 - `int` or `long` : integers
 - `float` or `double` : decimal numbers
 - `char` : characters
- Specifying data types tells the compiler: "reserve __ bytes of memory for this data when function is running!".
- On modern PCs (and most smartphones):
 - `int` and `float` occupies 4 **bytes** of memory.
 - `long` and `double` occupies 8 **bytes** of memory.
 - `char` occupies 1 byte of memory.

Declarations

- Variable in C is a placeholder of some value.
- The value held by a variable can be changed later.
- In C programming language, all variables must be declared.
- The syntax of declaration is: `data_type variable_name`.
 - Declaration: `double gravity;`
 - Declaration with initializations: `double m1 = 1.0;`
 - Declaration of multiple variables of the same type:
`double m1, m2; .`
 - Declaration of multiple variables of the same type with initializations: `double m1 = 1.0, m2 = 2.0; .`

Flow Control

If-Else

- This simple conditional statement is called "if-else" statement and exists in many programming languages.
- If-Else statement in C is written as follows:

```
if (condition){  
    statements to be executed,  
    if condition is true  
    ...  
} else{  
    statements to be executed,  
    if condition is false  
}
```

- If condition is true, the program bypasses all statements following `else`.
- `condition` can be logical and relational expressions.

Relational and Logical Operators

- Relational Operators

- `>` strictly greater. `2>1` is TRUE.
- `>=` greater or equal. `2>=3` is FALSE.
- `==` equals to. `1 == 1` is TRUE.
 - Note, single `=` is assignment operator. It assigns the value of RHS to the variable on the LHS. Do not get confused!
- `!=` not equal. `2 != 1` is TRUE.

- Logical Operators

- `&&` logic AND. `1>0 && 1>-1` is TRUE.
- `||` logic OR. `1>0 || -1>0` is TRUE.

if-else-if Ladder

```
if (condition1){  
    statment1;  
} else if(condition2){  
    statment2;  
} else if(condition3){  
    statment3;  
}  
...  
else{ //optional  
    statement0;  
}
```

- The program will check conditions **sequentially**.
- Once a true condition is found
 - It executes the associated statements.
 - then **bypasses the rest of the ladder**.
- If none of the conditions are true, it executes the `else` statements (if there is one).

While Loop

The simplest loop is while-loop and its syntax is:

```
while(condition){  
    statements  
}
```

The statements inside of the brackets will be run repeatedly as long as the `condition` is true.

```
// print out every positive integer smaller or equal than 10  
int i = 1;  
while(i<=10){  
    printf("%d\n", i);  
    i = i + 1;  
}
```

Note that `i` changes every iteration.

For Loop

- For loop is another type of loop mechanism in C.

```
for(init statement; condition; update statement){  
    statements to be repeated  
}
```

- i. It initializes a counter.
- ii. Check condition,
 - If it is satisfied, run statements
 - If not, exit the loop.
- iii. Run update statement. Go back to ii.

Nested Flow Control

Nested If-Else

- You can write conditional statement inside another conditional statement.

```
if(score >= 40){  
    printf("congratulations! ");  
    if(score >=70){  
        printf("first class!\n");  
    }else{  
        printf("passed!\n");  
    }  
}else{  
    printf("student has failed!\n");  
}
```

- The code prints out
 - student has failed! if score < 40 .
 - congratulations! passed! if 40 <=score < 70 .
 - congratulations! first class! if score >= 70 .

Nested Loops

- Similarly, you can write one loop inside another loop.

```
for (int i = 1; i <= 4; i=i+1){  
    // print i-th line  
    for (int j = 1; j <= 4; j=j+1){  
        printf("*");  
    }  
    printf("\n"); // change line  
}
```

- It prints out a block of *

```
****  
****  
****  
****
```

Early Loop Exit

- `break;` , early loop exit.
- `continue;` , early loop restart.

return Statement

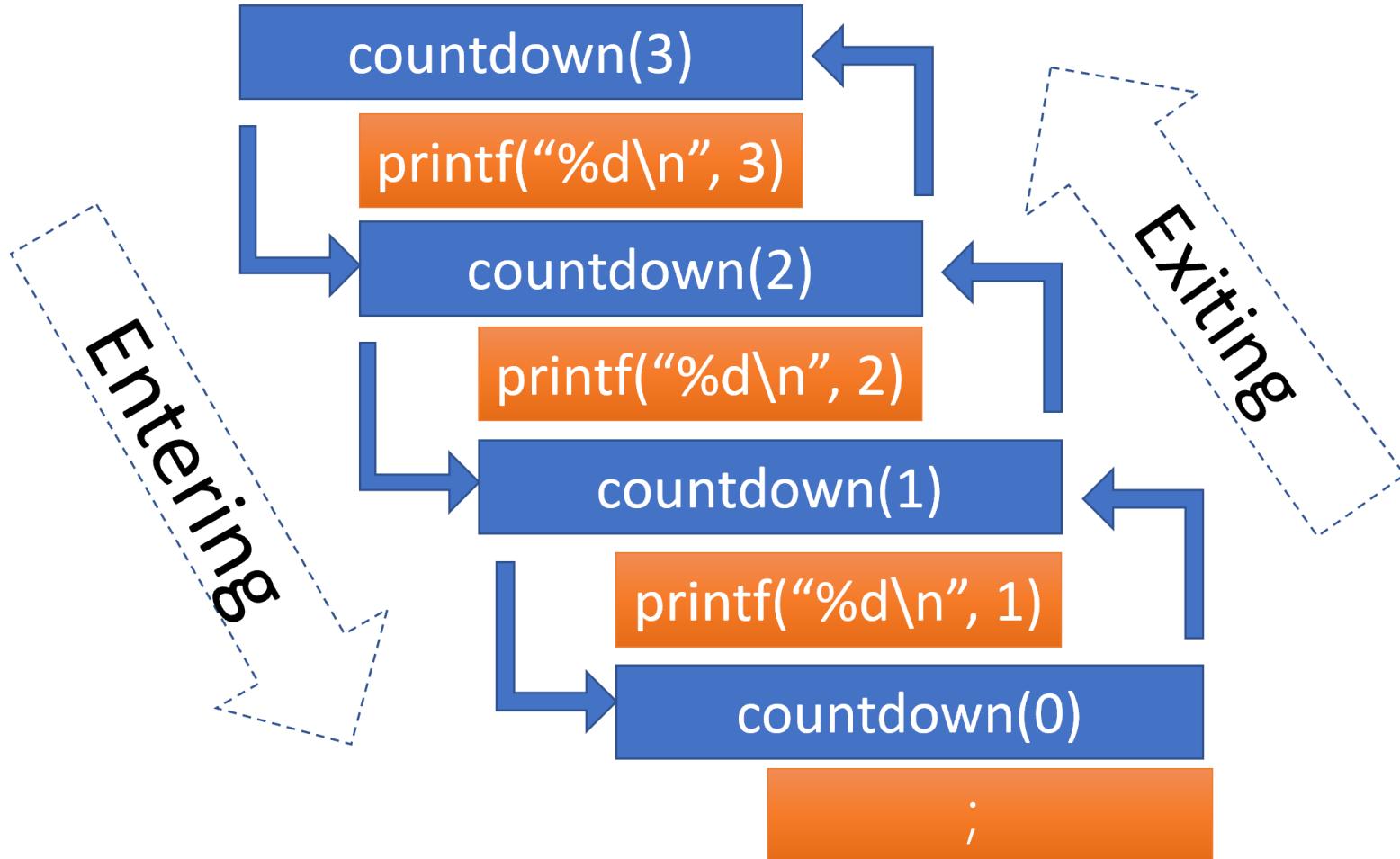
`return expression;`, will return the value of `expression`, and exit the function immediately.

Recursion and Stack Memory

Recursion

- You **cannot** define a function inside another function.
- You can **call** a function inside another function.
 - A function can call itself!

How does Recursion Work?



Memory Allocation for Functions

- When the function is being executed on the CPU, its data (such as variables declared in the function) and code are temporarily stored in the memory.
- The memory region for storing function data in the current program is called "stack".

Stack

- Stack is a data structure that the newest element is placed on top of the old elements.



Stack Memory Allocation

1. When a function is called, its data and code is added to the top of the stack.
2. CPU can only access data and code from **the top stack**.
 - It means, CPU can only access variables from the current function that is being called!
3. When a function finishes its execution, its data is removed from the stack and the space it occupies is freed for future calls of functions.

Array

Array

- Array is a fundamental **data structure** in C programming language that **stores a sequence of elements**.
- You can declare an array using the syntax:

- `data_type variable_name[array_size];` .
- `// declares an int array with 100 elements.`
`int a[100];`

- `array_size` can **NOT** be a variable.

- `int c = 100;`
`int b[c]; // compilation error!`

- `array_size` must be determined at the time of compilation.

Accessing Array

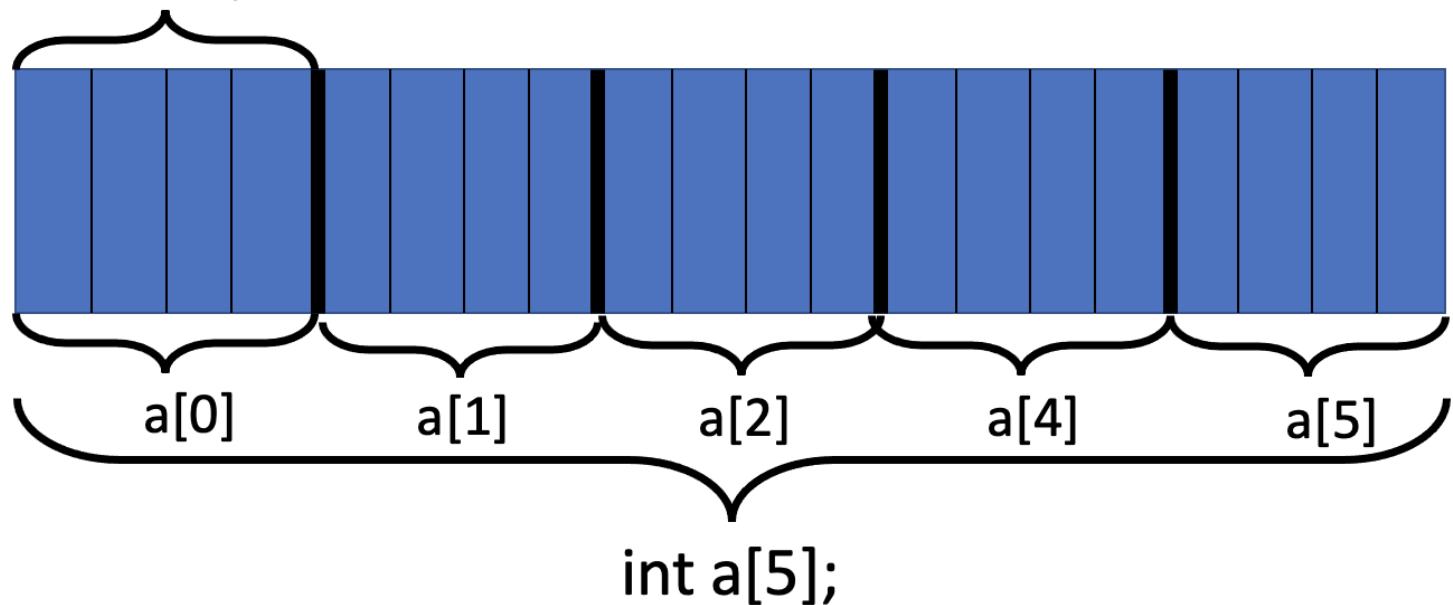
- The first element in the array is `a[0]`.
- The second element in the array is `a[1]`.
- and so on.
- e.g., `a[2] = 5;` assigns 5 to the third element of `a`.
- You can access multiple elements using a loop:

```
int a[10];
for(int i = 0; i < 10; i = i+1){
    a[i] = 123; // assigning the i+1-th element
}
for(int i = 0; i < 10; i = i+1){
    printf("%d ", a[i]); // print the i+1-th element
}
```

Array's Memory Layout

- Array is stored in a **contiguous section memory**.

$\text{int} = 4 \text{ bytes}$



Passing by Value and Passing by Reference

1. When calling a function, variables are passed to the function by creating a copy of that variable. Hence, the variable is passed by "value".
2. However, arrays are passed by reference, meaning no copies are made when passing the array.
 - i. We know why in the next lecture!

Pointers

Pointer

- Pointer is a variable that **stores the address of another variable in the virtual memory.**
- Using a pointer we can access the content stored in that memory location.
 - Like index card to a book in the library.

Declare and Initialize Pointer

- A pointer itself is a variable in C, thus should be declared before use.
 - Syntax: `data_type *var_name;`
 - For example,
 - `int *pa;` declares a `int` pointer
 - `double *pb;` declares a `double` pointer.
 - `int *pa = &a;` initialize `pa` with the address of `a`.
 - `&` is the "address of" operator. `&a` gives you the virtual memory address of `a`.

& Operator

```
#include <stdio.h>
void main(){
    int a = 0;
    printf("The address of a is %p.\n", &a);
    // displays "The address of a
    // is <some memory address>".
}
```

- Use `%p` to print out a pointer.

* Operator

- * is the opposite of & , called dereference operator.
- * takes the value from a certain memory address.

```
#include <stdio.h>
void main(){
    //initialize the pointer to be the address of a.
    int a = 1;
    int *pa = &a;
    int b = *pa;
    printf("My value is %d.\n", b);
    // displays "My value is 1."
}
```

Pointer and Array

In fact, in C, array name is a pointer pointing to the first element of the array!!

```
#include <stdio.h>
void main(){
    int a[3] = {2,3,4};
    int *pa = &a[0];
    printf("%p\n", a);
    printf("%p\n", pa);
    //prints out
    // 000000c0db3ffbec,
    // 000000c0db3ffbec, the same thing.
}
```

Dynamic Memory Allocation

Stack Memory: Pros & Cons

Stack is a highly efficient memory allocation/release mechanism.

- You do not need to free up stack memory used by your program.
 - Operating system (OS) cleans it up for you.
- However, the amount of stack memory must be determined at the compilation time!
 - OS must know how much stack memory your program uses before your program runs!
 - Allocating memory at compilation time is called **static memory allocation**.

Dynamic Memory Allocation

- We need a mechanism to **dynamically allocate memory spaces** for variables whose sizes cannot be determined before compilation.
- Memory allocation at **runtime** is called **Dynamic Memory Allocation**.
- In C programming language, variables that require dynamic memory allocation are stored in the **heap memory** (which is a part of the virtual memory).

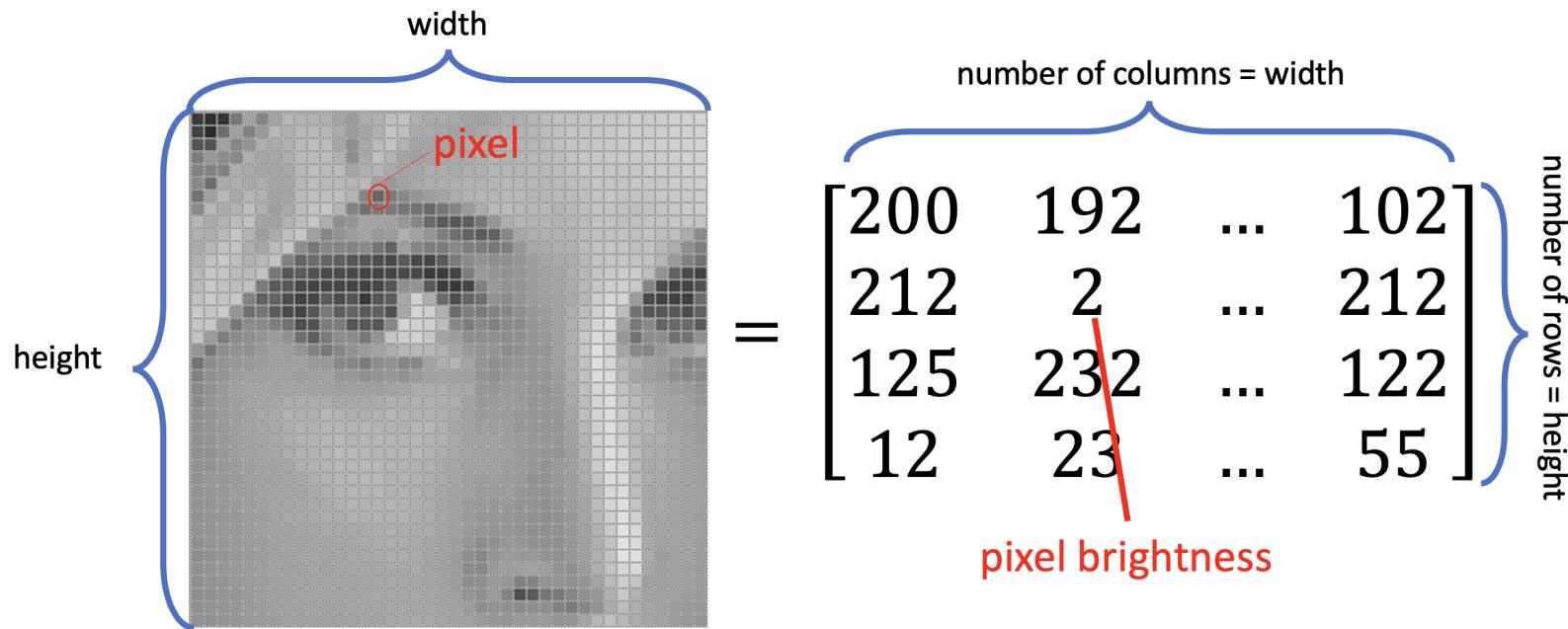
Heap Memory: Pros & Cons

- Your program can allocate heap memory to store variables while it is running.
 - The size of the allocated memory does **not** have to be known before compilation.
- However, you have to **manually allocate and free heap memory**.
 - Allocate Heap Memory, `malloc` .
 - Free Heap Memory, `free` .
 - They are provided in the **header file** `stdlib.h` .

Heap Memory Management

- `malloc`
- `free`
- `calloc`
- `realloc`

Images are Matrices



- Grayscale images are expressed as matrices in computer.
- A pixel in the image corresponds to an element in the matrix.
- Each element of the matrix indicate the brightness of a pixel. There are usually 256 levels of brightness, 0 is darkest while 255 is brightest.

Structure and Basic File IO

Row Major and Column Major

- Row Major and Column Major are two methods storing a matrix in an array.
- Matrix is a "2D object", you need to flatten it before storing it in a sequential container (such as an array).
- Use zero-based indexing (indices i, j starts from 0),
- Row-major order stores a matrix as

$$A = \begin{bmatrix} A_{00}, & A_{01}, & A_{02} \\ A_{10}, & A_{11}, & A_{12} \end{bmatrix} \implies [A_{00}, A_{01}, A_{02}, A_{10}, A_{11}, A_{12}]$$

- Row major order means $A_{i,j}$ is the `i*ncol + j`-th element in the array.

Grouping Variables

- Since `len1, vec1` are all variables describing the vector, we can group them together.
- Introducing a C language feature: **Structure**.
- A structure groups several related variables into a single entity.

Structure

- Syntax for defining a structure:

```
struct structure_name{  
    data_type variable1;  
    data_type variable2;  
    ...  
};
```

- Do not forget the ; at the end!
- Syntax for declaring a structured variable
- Syntax for referencing a sub-level variable contained in a structure variable

```
struct structure_name struct_variablename;
```

```
struct_variablename.variable1
```

Examples:

- Student example
- Vector example

File IOs

- `fopen`
- `fclose`
- `fgetc`
- `fputc`
- `fprintf`
- `fgets`
- `feof`