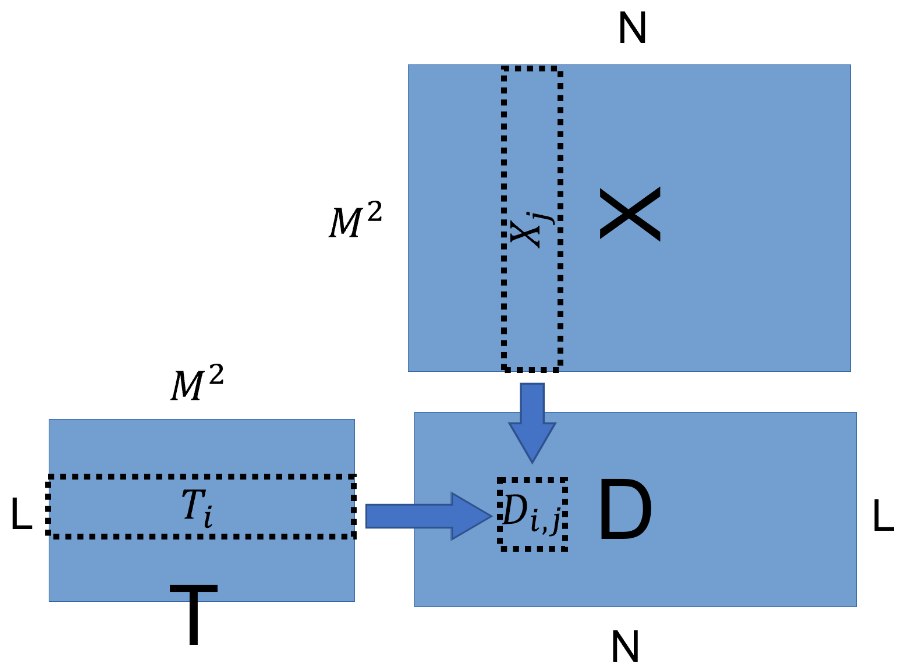
Part III of CW2

Given: A distance matrix $D \in \mathbb{R}^{100 \times 1987}$, and a label vector $Y \in \{0, 1, 2, \ldots, 9\}^{1987 \times 1}$.

- 

Target: find the 5 images that are "closest" to the the testing image i, and use their labels to predict the label of test image i.

Part III of CW2

Target: find the 5 images that are "closest" to the the testing image i, and use their labels to predict the label of test image i.

Divide and Conquer:

1. Obtain the indices of 5 smallest elements at row D_i.
2. Obtain the labels associated with these indices from Y.
3. Count number of "1"s, and check if the count is >= 3.

Finding 5 Smallest Elements

Let us focus on task 1:

How do you find 5 smallest elements in an array?

The Straightforward Approach

You can

1. Find the smallest element.
2. Find the second smallest element.
3. Find the third smallest element …

The Straightforward Approach

Pros:

- Easy to code, easy to understand.

Cons:

- A lot of similar code: Finding the smallest elem. is similar to finding the second smallest and is similar to ...

Writing similar code many times is bad in programming: If you make changes to one of them and forget to change the others, you have introduced a bug.

The Alternative Approach

1. Find the smallest element.
2. Set the smallest element to a huge number (`INT_MAX`).
3. Find the smallest element.
4. Set the smallest element to a huge number (`INT_MAX`).

   ... repeat 5 times.

Task 1 and 3 are the same! Write it into a function and call this function over and over again!

The Alternative Approach

```
int find_min_index(int a[], int len){
    // ...
}

void minimum5(int a[], int len, int indices[])
{
    for (int i = 0; i < 5; i++)
    {
        indices[i] = find_min_index(a, len);
        a[indices[i]] = INT_MAX;
    }
}
```

The Alternative Approach

Pros:

No repetitive code! Modular design.

Cons:

Slightly less straightforward.

- When writing a program, you have to balance between readability and a better design.
- Sometimes, it worth pursuing a "cleaner code" and forgo some readability.
- Sometimes, it worth making the code more readable, at the cost of slightly messier code.

Sort

Song Liu (song.liu@bristol.ac.uk)
GA 18, Fry Building,
Microsoft Teams (search "song liu").

Sorting

- The task of rearranging **items** in a **container** according to a pre-specified **order**.
  - Items: Numbers, Text, Dates...
  - Containers: Arrays, Lists, Tables...
  - Order: Numerical, Alphabetical, Chronological...
- Sorting is a frequently encountered task in computing.
  - Rank students by their grades.
  - Rank webpages by their relevance to user's search keywords.
- Today, we only talk about **sorting numbers in an array**, in **ascending order**.

Sorting

- Usually, sort algorithms are already implemented as a part of the programming language.
- In Python, you can do:
- ```
  >>> a = [3,5,4,2]
  >>> a.sort()
  >>> a
  [2, 3, 4, 5]
  >>> a = ['song','bob','anthony']
  >>>a.sort()
  >>> a
  ['anthony', 'bob', 'song']
  ```
- There are many existing sorting algorithms.
  ```

Preparation

- For our convenience, let us write a function that swap two elements in an array.
- ```
  void swap(int array[], int i, int j){
      int temp = array[i];
      array[i] = array[j];
      array[j] = temp;
  }
  ```
- ```
  int a[] = {1,2,3,4};
  swap(a, 1, 3);
  // Now, a = {1, 4, 3, 2}
  ```

Preparation

- Moreover, let us write a function that finds the index of the maximum element in an array with length `len`.
- ```
  int find_max_index(int array[], int len){
      int idx = 0;
      int max = array[0];
      for (int i = 1; i < len; i++)
      {
          if(array[i] > max){
              max = array[i];
              idx = i;
          }
      }

      return idx;
  }
  ```
  ```

Preparation

```c
int a[] = {2,4,3,1};
int max_idx = find_max_index(a, 4);
// max_idx is 1.
```

How do you sort?

- We know how to find the 5 smallest elements in an array.
- Imaging an algorithm finding the n smallest elements in a length n array.
    i. Find the smallest,
    ii. Find the second smallest,
    iii. ...
    iv. Find the n-th smallest,
- If we collect all the output from previous steps and put them into a **new array**, we should have a sorted array in ascending order.
- Can we sort without creating a new array?

A Sorting Algorithm

Assume you are sorting an array in ascending order.

1. Find the largest in [3,4,5,2], put it at the end.
    - [3,4,2,5].
2. Find the 2nd largest in [3,4,2,5], put it at the second last.
    - [3,2,4,5].
3. Find the 3nd largest in [3,2,4,5], put it it at the third last.
    - [2,3,4,5].
4. Done.

Find Pattern!

0. [3,4,5,2]
1. [3,4,2,5]
2. [3,2,4,5]
3. [2,3,4,5]

Find Pattern!

0. [3,4,5,2]
1. [3,4,2,5]
2. [3,2,4,5]
3. [2,3,4,5]

At step `i`, I actually swapped the `i`-th largest element with `a[len-i]` in the array.

- At step 1, I swapped the 1st largest with `a[len-1]`.
- At step 2, I swapped the 2nd largest with `a[len-2]`.
- At step 3, I swapped the 3rd largest with `a[len-3]`.

Pseudo Code, 1.0

- Input: Array `a` with length `len`.
- Output: Array `a` following the ascending order.

1. Finding the largest element in `a`.
    - Swap the 1st largest with `a[len-1]`.
2. Finding the 2nd largest element in `a`.
    - Swap the 2nd largest with `a[len-2]`.
3. Finding the 3rd largest element in `a`.
    - Swap the 3rd largest with `a[len-3]`.

    ...

`len-1`. Finding the `len-1` th largest element in `a`.

- Swap the `len-1` th largest element with `a[len-(len-1)]`.

Find Pattern!

0. [3,4,5,2]
1. [3,4,2,5]
2. [3,2,4,5]
3. [2,3,4,5]
4. [2,3,4,5]

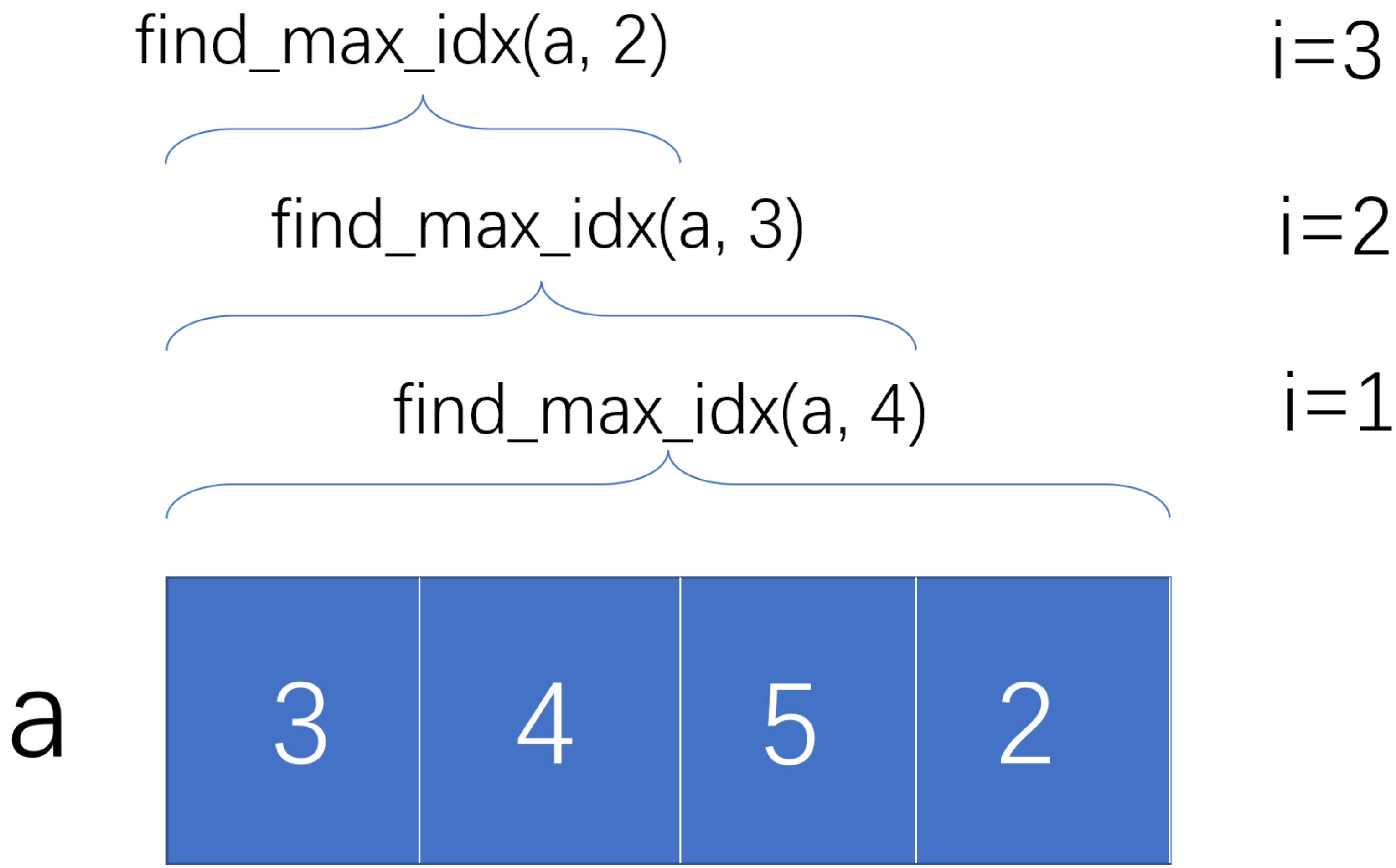
- Notice that at step `i`, last `i` elements are all sorted.
  - After step 1, `a[3]` to `a[3]` are sorted.
  - After step 2, `a[2]` to `a[3]` are sorted.
  - After step 3, `a[1]` to `a[3]` are sorted.
  - After step 4, `a[0]` to `a[3]` are sorted.

Pseudo Code, 1.0

- Input: Array `a` with length `len`.
- Output: Array `a` following the ascending order.
- For i from 1 to len-1
  - // find maximum from the **unsorted part** of the array,
  - // and swap it with the `i`-th last element
  - `max_idx = find_max_idx(a, len - i + 1)`
  - `swap(a, max_idx, len-i)`;

Homework: why `find_max_idx(a, len - i + 1)` finds the maximum from the unsorted part of the array?

Pseudo Code, 1.0



Sort 1.0

```
void sort(int len, int array[]){
    for (int i = 1; i <= len-1; i++){

        int max_idx = find_max_idx(array, len - i + 1);

        swap(array, max_idx, len - i);
    }
}
```

Example:

```
int a[] = {5, 3, 2, 1, 2,  4};
sort(a, 6);
```

after each swap, array looks like

```
Iteration 1: 4 3 2 1 2 5
Iteration 2: 2 3 2 1 4 5
Iteration 3: 2 1 2 3 4 5
Iteration 4: 2 1 2 3 4 5
Iteration 5: 1 2 2 3 4 5
```

Find Pattern!

0. [3,4,5,2]
1. [3,4,2,5]
2. [3,2,4,5]
3. [2,3,4,5]
4. [2,3,4,5]

- Notice that at step `i`, after swapping elements,
- We only need to apply **the exact same operation** to **the subarray** `a[0]` to `a[len-i-1]`.
- We can do this by **recursion**.

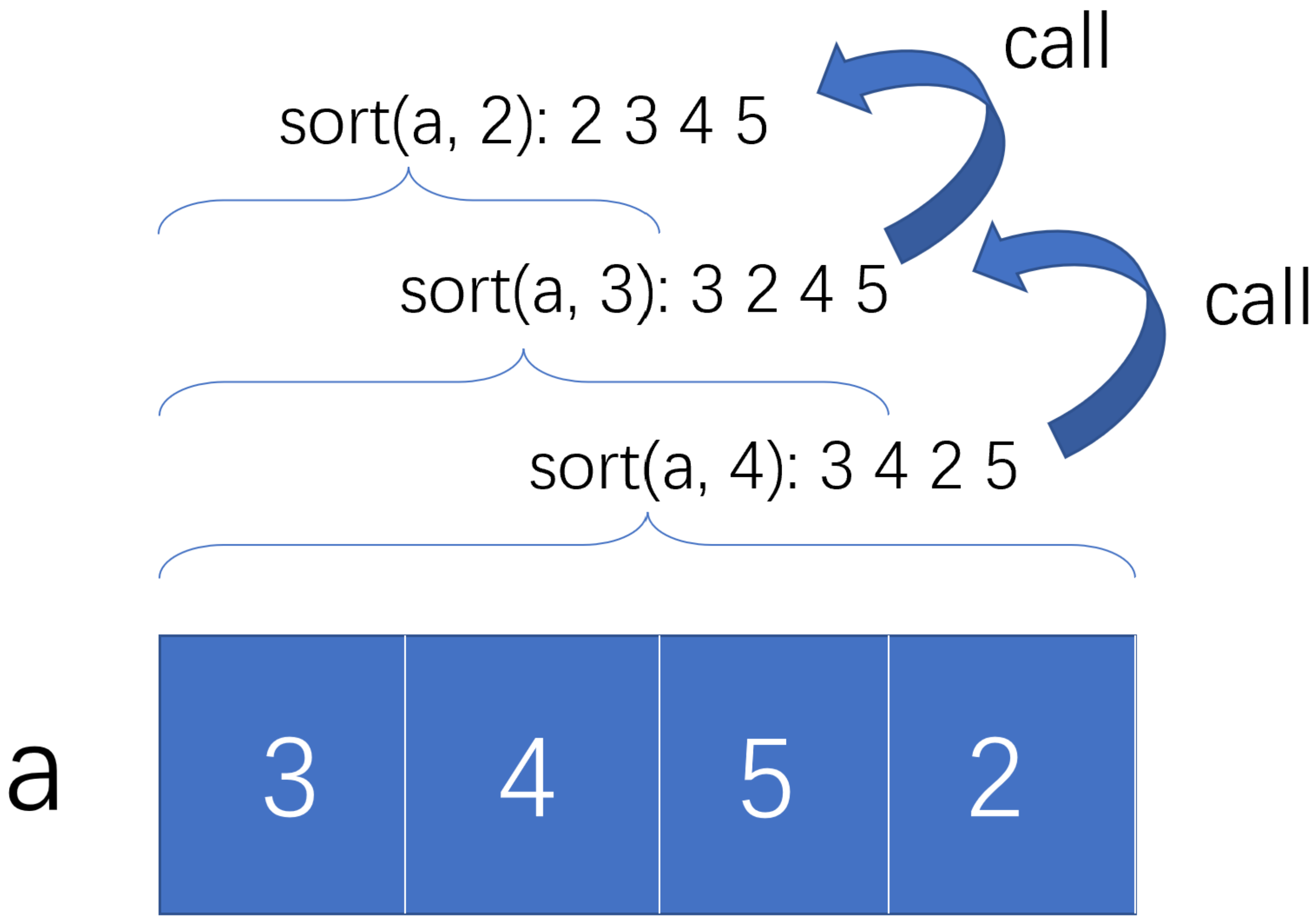
Pseudo Code 2.0

- Input: Array `a` with length `len`.
- Output: Array `a` following the ascending order.

Sort:

1. Find the index of the maximum element.
2. Swap the maximum with the `len-1`-th element (the last element in the **subarray**).
3. If `len` > 1
    - Sort rest of the array by calling `sort(a, len - 1)`.

Pseudo Code 2.0



Sort 2.0

```
void sort_v2(int array[], int len){
    int max_idx = find_max_idx(array, len);
    swap(array, max_idx, len - 1);

    if(len>1){
        sort_v2(array, len-1);
    }
}
```

Sort 2.0

```
int a[] = {5, 3, 2, 1, 2,  4};
sort(a, 6);
```

after each swap, array looks like

```
Iteration 1: 4 3 2 1 2 5
Iteration 2: 2 3 2 1 4 5
Iteration 3: 2 1 2 3 4 5
Iteration 4: 2 1 2 3 4 5
Iteration 5: 1 2 2 3 4 5
```

Sort 1.0 and Sort 2.0 are the same algorithm with different implementations, i.e., Loop vs. Recursion.

Time Complexity

- Clearly, as the length of the array gets longer, our sorting algorithm is getting slower.
- **How much slower?**
- We need to call `find_max_idx` `len-1` times.
- Each time, `find_max_idx` loop over `len-i+1` elements.
  - In total, we have `len` + `len-1` + `len-2` ... + `2` `for` loop iterations.
  - That is $\frac{(\text{len}+2)(\text{len}-1)}{2}$ loop iterations.
  - It grows quadratically with `len`!

Time Complexity

- That means, if the sorting algorithm takes t seconds to run on a 1000 elements array.
- It is likely to take $4t$ seconds to run on an 2000 element array.
  - When `len` is large, quadratic term dominates.
- We can say our sorting algorithm has a time complexity $O(\mathrm{len}^2)$.
  - We only care the dominating term as `len` increases.
- Time complexity describes how computational time grows with the problem size.

Time Complexity

- The time complexity of printing an length `n` array: $O(n)$.
- The time complexity of printing an $m \times n$ matrix: $O(mn)$.
- The time complexity of computing the multiplication between an $m \times k$ matrix and a $k \times n$ matrix : $O(???)$.

Time Complexity

- Given a problem size n, we can divide algorithms into several categories based on their time complexities:
- Constant time: $O(1)$.
- Linear time: $O(n)$.
- Quadratic time: $O(n^2)$.
- Exponential time: $O(2^n)$.

These complexities indicate the how hard a problem is, as size n increases.

- Computational complexity plays a central role in one of the biggest unsolved Mathematical mystery.
- Watch this Youtube video if you are interested.

Conclusion

- Sort: put elements in the array in order.
  - For loop version
  - Recursive version
- Time Complexity: How the computational time grows with the problem size.

Homework: Sort

1. Read lecture slides, write functions
   `int find_max_idx(int array[], int len)`
   and
   - In the `find_max_idx`, adding a line to print out the following message:
     \- `The maximum is %d`, replace `%d` with the maximum.

Homework: Sort

2. `void swap(int a[], int i, int j)`.
    - In the `swap` function, adding a line of code to print out the following message:
    - `I am swapping %d with %d`, replace `%d` with elements to be swapped.
3. Write test cases for your implementation of `find_max_idx` and `swap`. Make sure they are implemented correctly.

Homework: Sort (submit)

3. Implement the sort algorithm (for loop) version.
    - Use the `swap` and `find_max_idx` you just wrote.
    - At each iteration,
    - Print out the array after the swap.
4. From the printed out generated by your code, see if the sort algorithm work as you imagined.

Homework: Sort (submit)

5. Assume your program runs on a slow computer. Each for loop iteration will take one second.
    - Suppose you are sorting a length 5 array. How many seconds will it take to run the sorting algorithm?
    - Suppose you are sorting a length 10 array. How many seconds will it take to run the sorting algorithm?
    - Write your answer in your submitted code, as a comment.

Homework: Selection Sort (submit)

6. The algorithm we introduced in the lecture is a simplfied verion of a more practical sorting algorithm, selection sort. Can you implement the selection sort algorithm based on the following pseudo code?

Homework: Selection Sort (submit)

Function Name: sort

Input: `a[len]`, an array. `len`, the array length.

Output: The sorted array `a[len]`.

```
sort(a, len)

    if len <= 1, return.

    for i from 0 to len-1
        if a[i] > a[len-1]
            swap a[i] and a[len-1]

    call sort(a, len - 1)
```

Homework: Selection Sort (submit)

Input: `a[len]`, an array. `len`, the array length.
Output: The sorted array `a[len]`.

```
for m from len-1 to 1
    for i from 0 to m-1
        if array[i] > array[m],
            swap array[i] and array[m].
```

Are these two algorithm more or less efficient than our method introduced in the lecture? Why?