Sorting

Matteo Fasiolo

About myself

Matteo Fasiolo (matteo.fasiolo@bristol.ac.uk)

GA 17, Fry Building,

Microsoft Teams (search "matteo fasiolo").

I have been programming mostly in R since 2006, I have authoured several R packages:

- mgcViz for visualising regression models
- qgam for non-parametric quantile regression
- mvnfast for fast computation with the multivariate normal distribution

I teach a PhD-level course one how to interface R with C++.

I do research on statistical modelling for energy applications.

Course organisation

Mostly as in TB1:

- Mon 10am lecture in PHYS BLDG B16/17 ENDERBY
- Tuesday 1pm lab in FRY BLDG LG.21 (note only 1 lab!)
- Tutorial every other week

Lab problem due the following Monday.

Two assessed courseworks (end of Feb and of April).

Course content will appear here:

- first 6 weeks spent on algorithms in C and C++.
- last 6 week spent on the R statistical language.

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**.

Usually, sort algorithms are already implemented as a part of the programming language.

In R, you can do:

```
a <- c(3,5,4,2)
sort(a)
2 3 4 5

a <- c('matteo','bob','anthony')
sort(a)
"anthony" "bob" "matteo"
```

There are many existing sorting algorithms.

Basic ingredients

For our convenience, let us write a function that swaps 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}
```

Let's write a function that finds index of max element in array:

```
int find_max_idx(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;
}
```

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

How do you sort?

How do we use the functions above to find the n largest elements in an n array?

Imagine something like this:

1. Find the largest,
2. Find the second largest,
3. ...
4. Find the n-th largest,

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? (sorting "in place")

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 at the third last.
   - [2,3,4,5].
4. Done.

Let the first step be the 0-th step, following the C convention.

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

That is:

- 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]`.

In general, at step `i` I swapped the `i`-th largest element with `a[len-i]` in the array.

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)]`.

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

Notice that at step `i`, last `i` elements are all sorted.

- After step 1, `a[3]` to `a[3]` are finalised.
- After step 2, `a[2]` to `a[3]` are finalised.
- After step 3, `a[1]` to `a[3]` are finalised -> done!

Input: Array `a` with length `len`.

Output: Array `a` following the ascending order.

For i from 1 to `len-1`

1. Find maximum from the **unsorted part** of the array with

   ```
   max_idx = find_max_idx(a, len - i + 1)
   ```

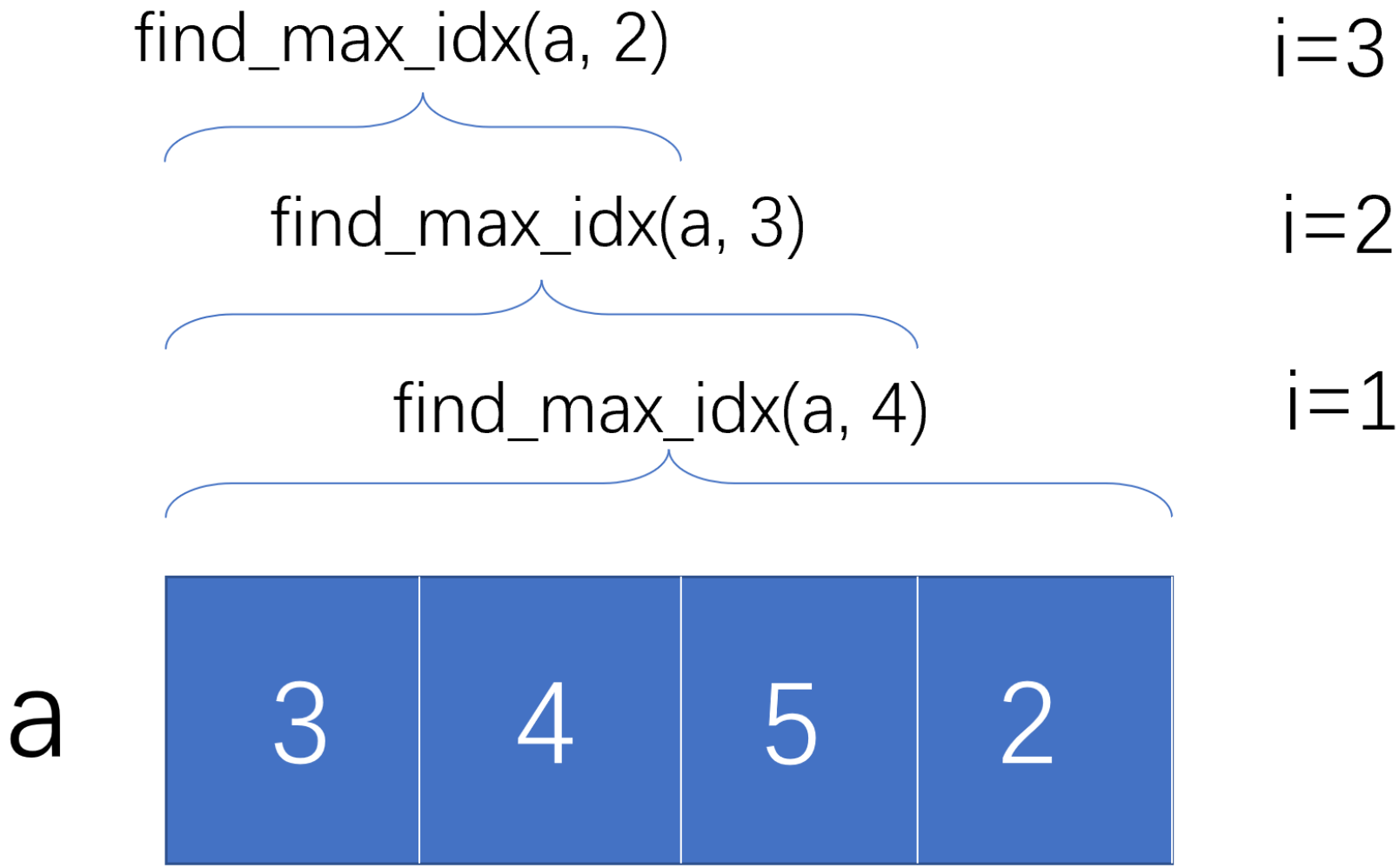
2. Swap it with the `i`-th last element using

   ```
   swap(a, max_idx, len-i)
   ```

Homework: why does

```
find_max_idx(a, len - i + 1)
```

find the maximum from the unsorted part of the array?


find_max_idx(a, 2)
i=3
find_max_idx(a, 3)
i=2
find_max_idx(a, 4)
i=1
a
3
4
5
2

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]

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 could do this by **recursion**.

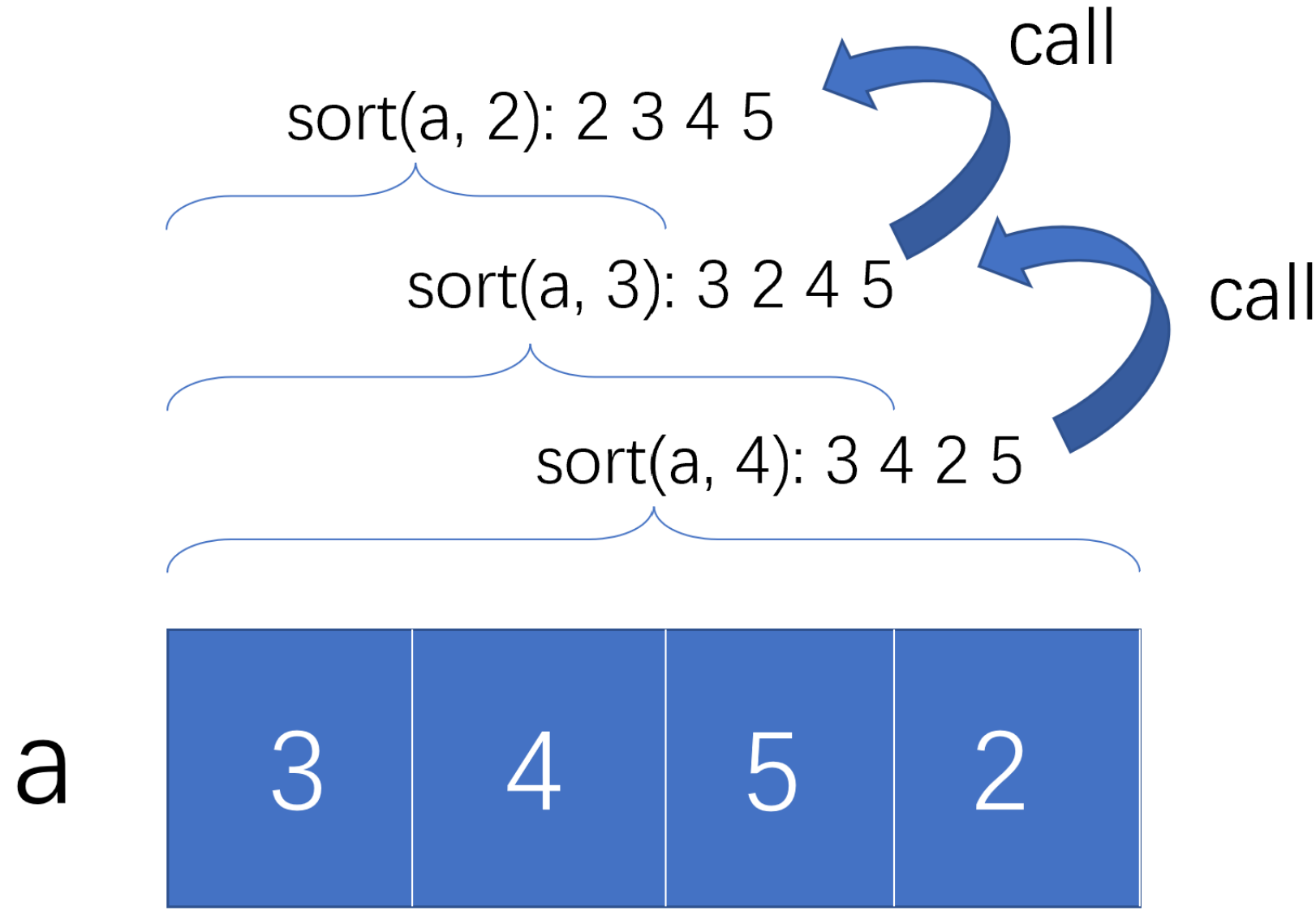
Pseudo Code 2.0

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)`.


call
sort(a, 2): 2 3 4 5
sort(a, 3): 3 2 4 5
call
sort(a, 4): 3 4 2 5
a
3
4
5
2

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);

    }

}
```

```
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.

At iteration `i`, `find_max_idx` loops over `len-i+1` elements.

- In total, we have `len` + `len-1` + `len-2` ... + `2` `for` loop iterations.
- That is $\frac{\text{len}(\text{len}+1)-2}{2} \propto \text{len}^2$ loop iterations.
- It grows quadratically with `len`!

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 $O(\text{len}^2)$.

- We only care the dominating term as `len` increases.

Examples:

- $\text{len} + \sqrt{\text{len}} + 1$ is $O(\text{len})$.
- $\log \text{len} + \frac{\text{len}^3}{2}$ is $O(\text{len}^3)$.

Time complexity describes how computational time grows with the problem size.

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 : $O(???)$.

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 mysteries.

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

See `lab_sort_template.c` file.

1. Read lecture slides, write functions
   - `int find_max_idx(int array[], int len)`
   - `void swap(int a[], int i, int j)`
2. In `find_max_idx`, add a line to print out the following message:
   - "`The largest element is %d`", replace `%d` with the largest element.
3. In `swap` add a line of code to print out the following message:
   - "`I am swapping %d with %d`", replace `%d` with elements to be swapped.
4. Write test cases for your implementation of `find_max_idx` and `swap`. Make sure they are implemented correctly.

Homework: Sort (submit)

5. 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.
6. From the printed text generated by your code, see if the sort algorithm work as you imagined.
7. 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?
   - Make sure that you code prints out the answer.

Homework: Selection Sort (submit)

8. The algorithm we introduced in the lecture is a simplified version of selection sort. Can you implement the selection sort algorithm based on the following pseudo code?

Input: array `a` and its length `len`

Output: The sorted array `a`.

```
sort_select_recur(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_select_recur(a, len - 1)
```

9. Implement the non-recursive implementation of selection sort.

Input: array `a` and its length `len`

Output: The sorted array `a`.

```
sort_select(a, len)

// Extra challenge: do not read the pseudo-code below!
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 algorithms more or less efficient than our method introduced in the lecture? Why?