Sort

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

GA 18, Fry Building,

Microsoft Teams (search "song liu").

A Simple Task

- Suppose you have an array of customer ratings.
- `int a[] = {3,4,5,2};`
- Sort the array in ascending order, so that after sorting, looks like `{2,3,4,5}`.
- Can you do it by hand?
    - Of course.

A Simple Task

- Perhaps you can do this task so efficiently, that you do not even notice how you have done it!
- Spend a few minutes, write down your sorting procedure.
    - Save your procedure, we will use it later.

Sorting

- The task of rearranging an array and putting its elements in order is called **sorting**.
- Sorting is a frequently encountered task in computer programming.
    - Rank students by their grades.
    - Rank webpages by their relevance to user's search keywords.
    - Schedule tasks to be run by the CPU based on their priorities.
- The sorting can be either descending or ascending, based on numerical or alphabetical order.

Sorting

- There are many existing sorting algorithms.
- Usually, some 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']
  ```
- Let us implement a sorting algorithm.
  ```

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}
  ```

We have practiced a similar task in Week 5 lab.

Preparation

- Moreover, let us write a function that finds the index of the maximum element in an array with length `len`.
  - Assuming elements in the array are bigger than -999.
- ```
  int find_maximum(int len, int array[len]){
      int min = -999;
      int idx = -1;
      for (int i = 0; i < len; i++)
      {
          if(array[i] > min){
              min = array[i];
              idx = i;
          }
      }

      return idx;
  }
  ```
  ```

Preparation

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

- We have practiced a similar task in Week 5 Lab.

How do you sort as a human?

Assume you are sorting an array in ascending order.

1. Find the largest element in the array.
2. Put it at the end.
3. Find the next largest element.
4. Put it before the largest element.
5. Find the next largest element.
6. Put it before the second largest element.

   ...

How do you sort as a human?

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 before the the largest element.
    - [3,2,4,5].
3. Find the 3nd largest in [3,2,4,5], put it before the the 2nd largest element.
    - [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 the `len-i+1`-th element in the array.

- At step 1, I swapped the 1st largest with the 4-th element.
- At step 2, I swapped the 2nd largest with the 3-rd element.
- At step 3, I swapped the 3nd largest with the 2-nd element.
- At step 4, I swapped the 4nd largest with the 1-st element.

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 biggest with the `len−1+1`-th element.
2. Finding the 2nd largest element in `a`.
    - Swap the 2nd biggest with the `len−2+1`-th element.
3. Finding the 3rd largest element in `a`.
    - Swap the 3rd biggest with the `len−3+1`-th element.

    ...

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

- Swap the `len−1`th largest element with the `len−(len−1)+1`-th element.

Find Pattern!

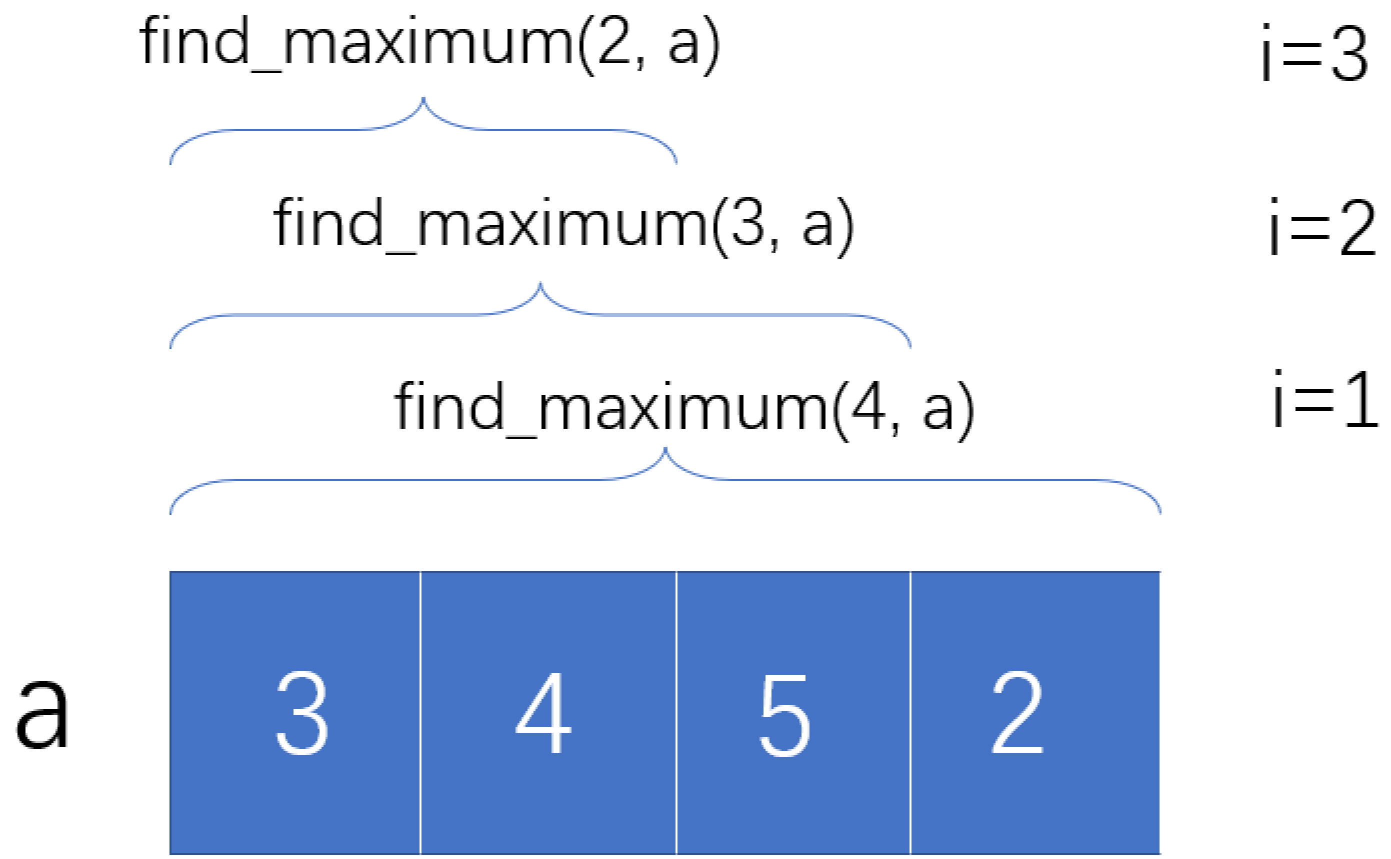
0. [3,4,5,2]
1. [3,4,2,5]
2. [3,2,4,5]
3. [2,3,4,5]
4. [2,3,4,5]

- Notice that after step `i`, elements from `a[len-i]` to `a[len-1]` 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,
  - // i.e., the first `len - i + 1` elements in the array.
  - `max_idx = find_maximum(len - i + 1, a)`
  - `swap(a, max_idx, len-i)`;

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_maximum(len - i + 1, array);
        swap(array, max_idx, len - i);
    }
}
```

Example:

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

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]

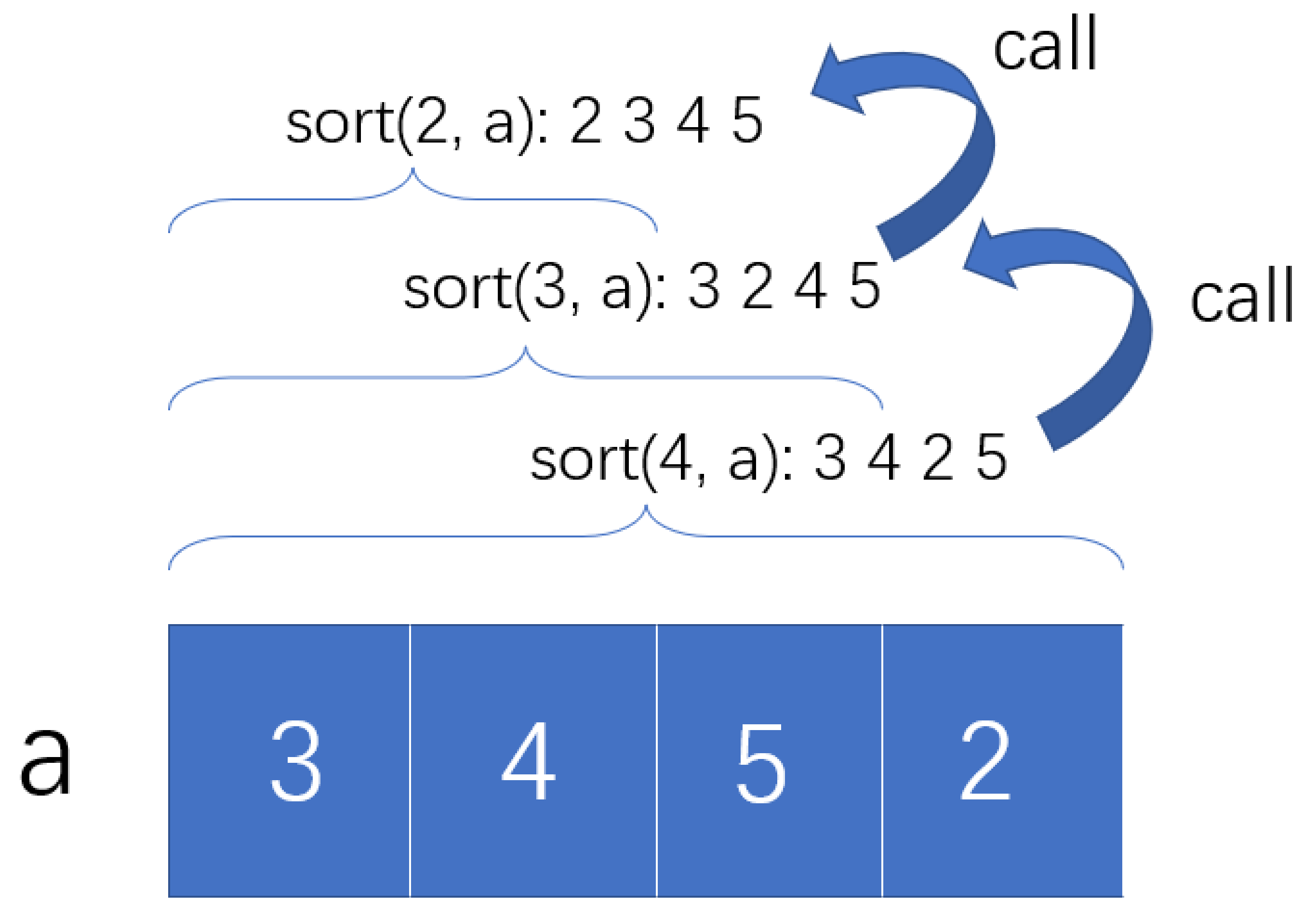
- Notice that after step `i`, elements from `a[len-i]` to `a[len-1]` contains the `i` biggest elements.
  - So, we only need to sort elements from `a[0]` to `a[len-i-1]`.
- We can do this by **recursion**.

Pseudo Code 2.0

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

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

Pseudo Code 2.0



Sort 2.0

```c
void sort(int len, int array[]){
    int max_idx = find_maximum(len, array);
    swap(array, max_idx, len - 1);

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

Sort 2.0

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

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.

Computational Complexity

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

Computational Complexity

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

Computational Complexity

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

Computational Complexity

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

Computational Complexity

- 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
- Computational Complexity: How the computational time grows with the problem size.