ICDS Spring 2025

Algorithms: Part I

Fundamentals and Sorting Algorithms

What will be involved in this topic?

Here, we consider algorithms as executable programs.

In lectures:

- understanding algorithms in a programer's perspective
  - How to represent an algorithm
  - How to convert an algorithm into runable code
- computational complexity
- algorithm design practice

In recitations:

- OOP syntax
- algorithm design exercises

Agenda

- What is an algorithm?
  - The definition of an algorithm
  - Representing an algorithm
  - Repetitive structures

- Sorting algorithms
  - Bubble sort
  - Merge sort

What is an algorithm?

An algorithm is a group of instructions.

- Everyday, we encounter various kinds of instructions.
  - recipes
  - traffic code
  - ....

But (strictly) they are not algorithms in computer science.

Definition of an algorithm in computer science

An algorithm is an **ordered** set of **executable** steps to accomplish a task.

- having a well-established structure in terms of the order of their execution
- steps must be executable
  - "make a list of all the positive integers" is **not** a executable step.
- steps must be **unambiguous**
  - "put a bit oil and salt on the steak" is **not** a precise step.
- the execution must lead to an **end**
  - (the description of) an algorithm **will stop** ultimately.

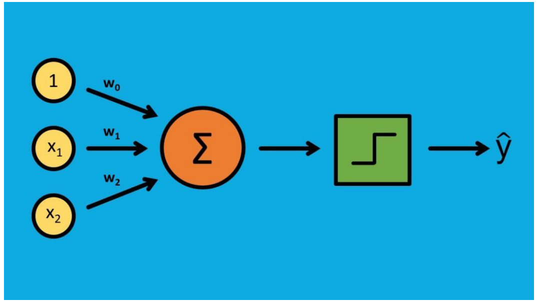
Algorithm 1 SLIC superpixel segmentation

/* *Initialization* */
Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S.
Move cluster centers to the lowest gradient position in a 3×3 neighborhood.
Set label $l(i) = -1$ for each pixel i.
Set distance $d(i) = \infty$ for each pixel i.
repeat
  /* *Assignment* */
  for each cluster center C_k **do**
    for each pixel i in a $2S \times 2S$ region around C_k **do**
      Compute the distance D between C_k and i.
      if $D < d(i)$ **then**
        set $d(i) = D$
        set $l(i) = k$
      end if
    end for
  end for
  /* *Update* */
  Compute new cluster centers.
  Compute residual error E.
until $E \leq$ threshold

Algorithm representation

An algorithm can be represented in many ways.

- pseudo-code,
- programming languages,
- natural languages, or even
- pictures
    - Flow chart



Pseudo-code

- Pseudocode is a **notational** system where **ideas** can be expressed **informally** during the algorithm development process.
- It is a personalized informal language. You can write your algorithm by mixing symbols and keywords in the way you like.
- Two principles to follow:
  - Being easier to understand than programming languages (i.e., often ignoring the implementation details)
  - Be more concise than natural languages (i.e., often using keywords and structures)

Algorithm 1 SLIC superpixel segmentation

/* *Initialization* */
Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S.
Move cluster centers to the lowest gradient position in a 3×3 neighborhood.
Set label $l(i) = -1$ for each pixel i.
Set distance $d(i) = \infty$ for each pixel i.
repeat keywords
  /* *Assignment* */
  for each cluster center C_k **do**
    for each pixel i in a $2S \times 2S$ region around C_k **do**
      Compute the distance D between C_k and i.
      if $D < d(i)$ **then** structures
        set $d(i) = D$
        set $l(i) = k$
      end if
    end for
  end for
  /* *Update* */
  Compute new cluster centers.
  Compute residual error E.
until $E \leq$ threshold

Anatomy of an algorithm

Algorithm is a **tangible form** of a solution to a problem.

Generally speaking, it is an executable program that often contains the following structures,

- Assignment
- Selection
- Repetition

Assignment: defining variable/function

The pseudocode "=" is directly follows the Python assignment statement.
It means storing a value into a variable.

For example, a = 3

Selection/Decision making

The truth value of the condition determines which activity to follow.

```
If condition:
   activity 1
else:
   activity 2
```

Selection structures can be nested:

```
If condition:
   activity 1
   if condition:
      activity 3
   else:
      activity 4
else:
   activity 2
```

Repetition

Algorithms are about what to repeat and when to repeat.

(Thanks to these structures! They free programmers from writing the same statements repeatedly.)

- Iterative structure (`for, while`)
  - A collection of instructions is repeated in a looping manner.

```
for n in nums:
    s += n
```

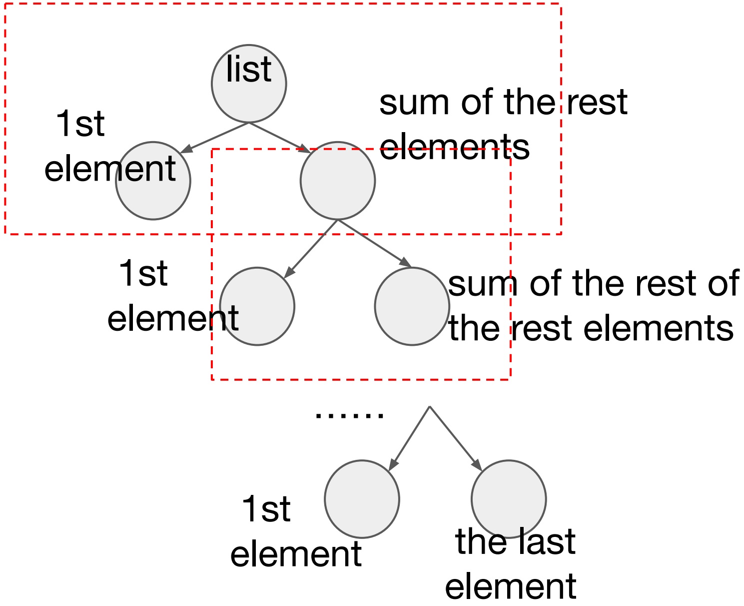
- Recursive structure
  - It involves repeating the set of instructions as a subtask of itself.

```
def sumup(lst):
    if len(lst)==1:
        return lst[0]
    else:
        return lst[0]+sumup(lst[1:])
```

Repetition is common

- Computation is a process that combines selection and repetition.

  e.g., summing all elements in a list ⇒ adding two numbers repeatedly



Note: Repetition is a **tool** for solving problems; You should design algorithms that adapt to the tool. ⇒ when you design algorithms, remember to ask yourself "How can I solved it using repetition?"

Recursion is a convenient way to implement repetition

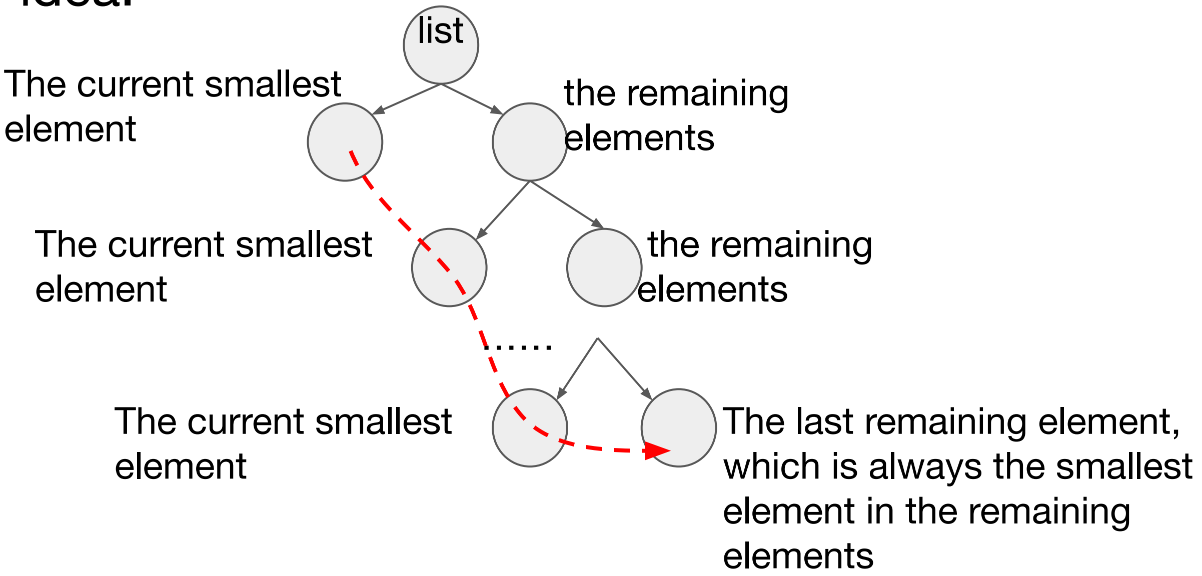
- It may look like cheating ⇒ We **define some value**, **pretending** that we **already know** it
  - e.g., the sum of the rest elements in the list is obtained by calling `sumup` itself

```
def sumup(lst):
    if len(lst)==1:
        return lst[0]
    else:
        return lst[0]+sumup(lst[1:])
```

- Recursion: a powerful and common technique in computer science
  - even in math ⇒ proof by induction is also based on this trick

Describe the algorithm to sort a list of numbers

Assume you are given a list [6, 5, 3, 1, 8, 7, 2, 4], how can you sort the numbers ascendingly? **Please thinking with recursion**, draw a tree diagram to show your idea.



Intuitive sort

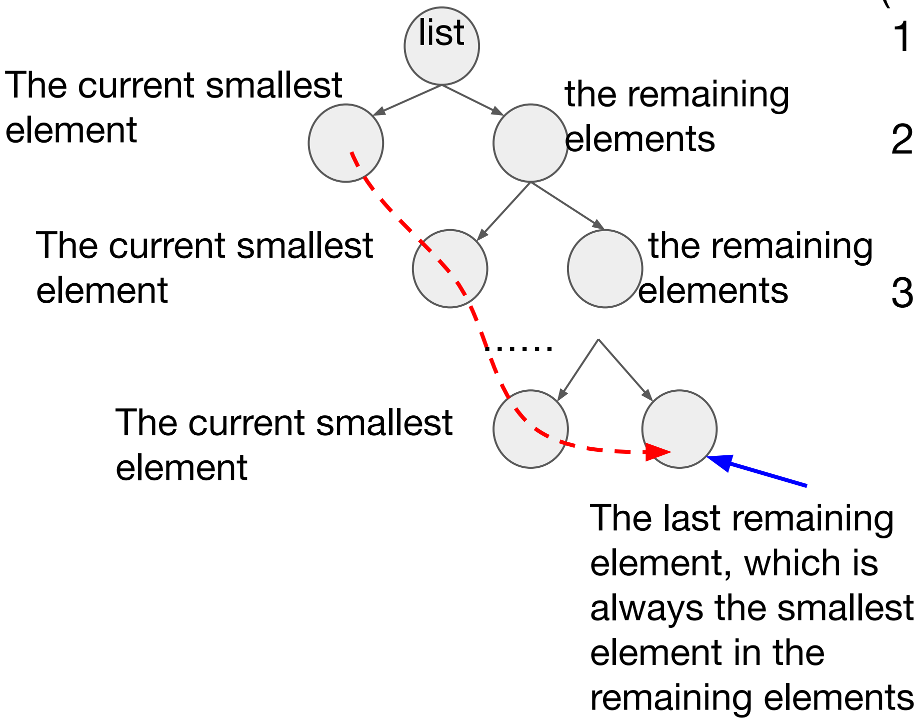
```
def find_min(unsorted_list):
    min_num = unsorted_list[0]
    idx = 0
    for i, num in enumerate(unsorted_list):
        if min_num > num:
            min_num = num
            idx = i
    return min_num, idx


def sort(unsorted_list):
    if len(unsorted_list) == 0:
        return unsorted_list
    min_num, idx = find_min(unsorted_list)
    unsorted_list.pop(idx)
    sorted_list = sort(unsorted_list)
    sorted_list.insert(0, min_num)
    return sorted_list
```

1. Find the smallest number in the unsorted numbers
2. Take it out
3. Sort the rest unsorted numbers
4. Insert the smallest number found in Step 1 in front of the sorted number in Step 3

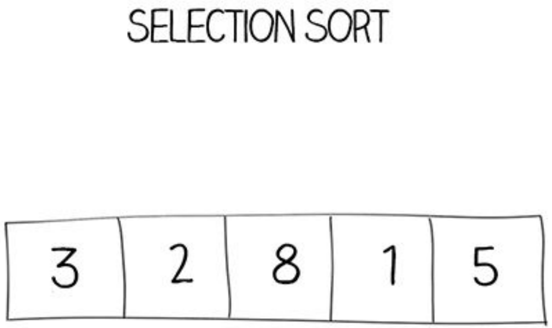
Sorting a list

Selection sort

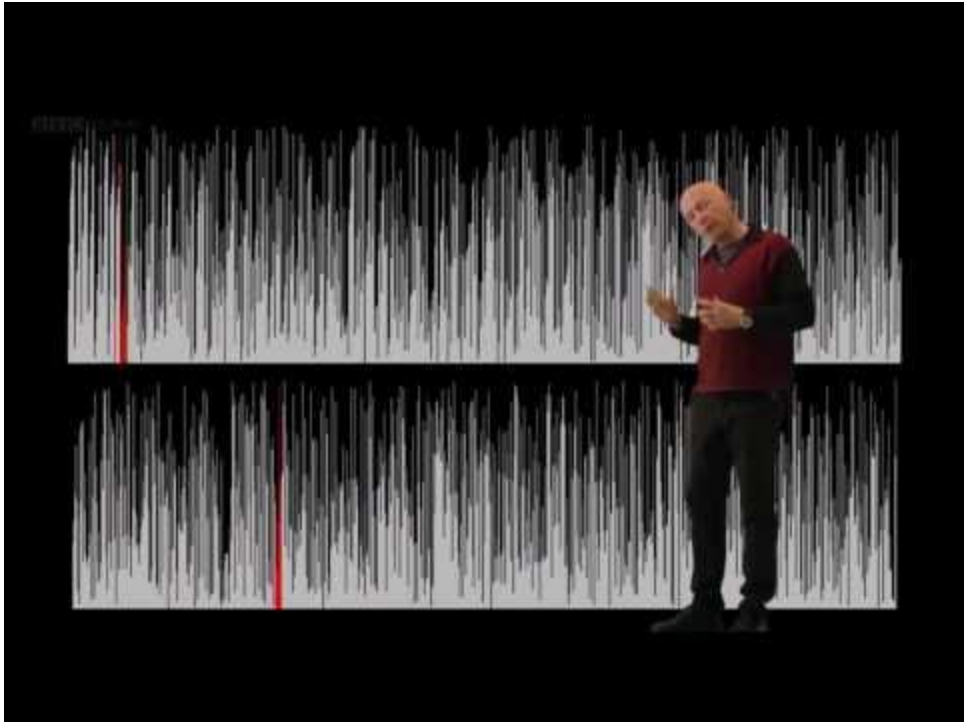


Let's follow the intuitive sort but move the numbers in-place; we have the **selection sort**: (You can implement it by for loops)

1. finding the minimum in the unsorted numbers
2. swapping it with the first number of the unsorted (so the minimum has been placed to the correct place)
3. repeat 1 and 2 until all numbers are sorted.



Bubble sort, another way to sort the list



How does this work?

6 5 3 1 8 7 2 4

Basic Idea:

- in a sorted list, neighbours are sorted (obvious)
- Parse the list and swap elements that are not sorted.
- Repeat until the list is sorted.

Bubble Sort: the basic idea

```
func bubble_sort(var a as array)
    for i from 1 to N:
        for j from 0 to N-1:
            if a[j] > a[j+1]:
                swap(a[j], a[j+1])

end func
```

Did we do bubble sort “efficiently”?

There are at least two obvious ways to improve the algorithm.

- Some pairs in the the given list are in the order already.
    - e.g., [0, 1, 4, 3, 6, 5, 7, 8]

- So, when shall we stop? Do we need to run N-1 passes always?

No, if there’s no swap, you can safely assume the list is sorted.

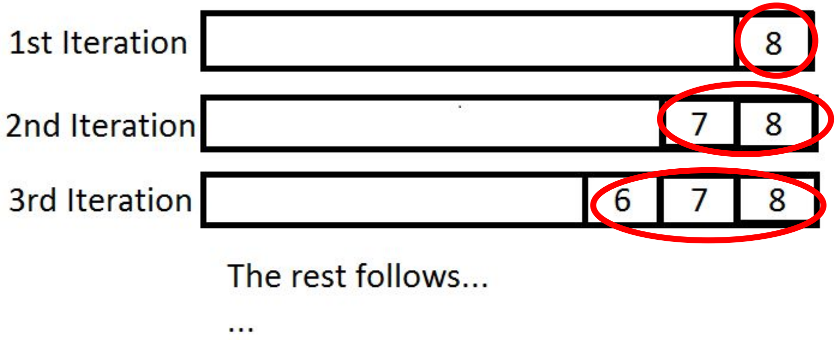
Bubble Sort: optimized

```
func bubble_sort2(var a as array)
    for i from 1 to N:
        swaps = 0
        for j from 0 to N-1:
            if a[j] > a[j+1]:
                swap(a[j], a[j+1])
                swaps = swaps + 1
        if swaps == 0
            break
end func
```

in each pass you keep track of whether or not any pair of elements was swapped;

Did we do bubble sort “efficiently”?

- Once we have placed the largest number at the end of the list, this number won’t be moved around anymore.



Bubble Sort: optimized

```
func bubble_sort3(var a as array)
    for i from 1 to N:
        swaps = 0
        for j from 0 to N-i:
            if a[j] > a[j+1]:
                swap(a[j], a[j+1])
                swaps = swaps + 1
        if swaps == 0
            break
end func
```

[Exercise] Implement a function that performs bubble sort on a list of numbers
Translate pseudocode into a function!

Any other ways to sort?

- We have a list which contains two sorted sublists:

1 3 7 9 2 4 6 8

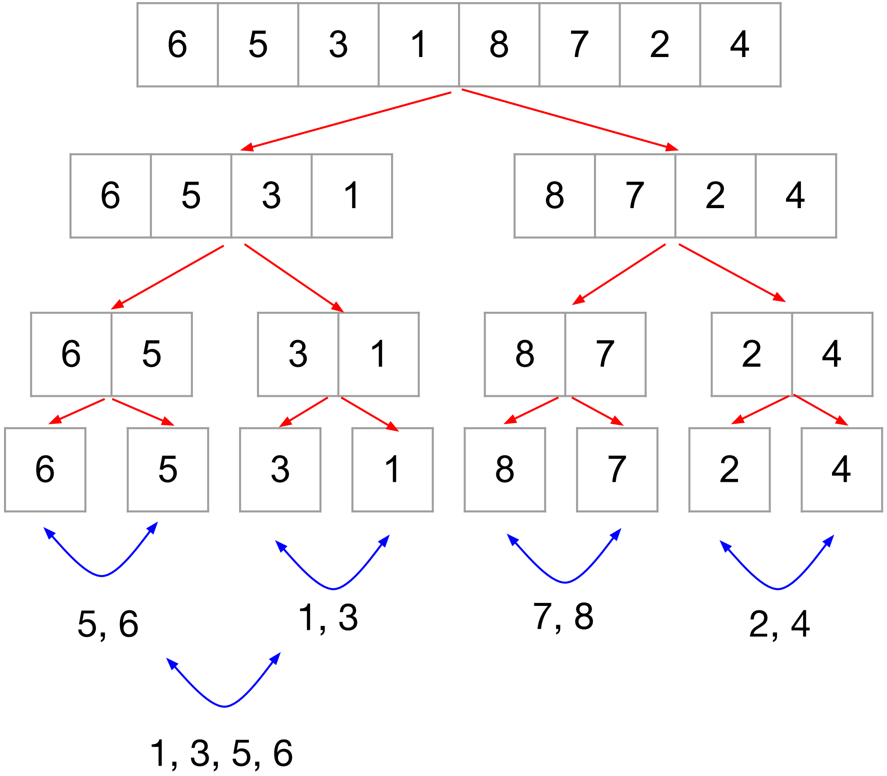
Can take use of the sorted part when sorting the list?

Yes, we can merge the sorted sublist!

Merge Sort: How does this work?

How does this work?

6 5 3 1 8 7 2 4




```python
# Merge sort definition
def merge_sort(m):
    if len(m) <= 1:
        return m
    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]
    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)
```

Reaching leaves

Find the nodes

Do something after getting the node values

Merge Sort: Pseudocode

```
func merge_sort(var a as array):
    if n == 1:   #n is the length of a
        return a
    var l1 as array = [a[0],...,a[n/2]]
    var l2 as array = [a[n/2+1],...,a[n]]
    l1 = merge_sort(l1)
    l2 = merge_sort(l2)
    return merge(l1, l2)
end func
```

```
func merge(var a as array, var b as array):
    var c as array
    while a and b have elements:
        if a[0] > b[0]:
            add b[0] to the end of c
            remove b[0] from b
        else:
            add a[0] to the end of c
            Remove a[0] from a
     while a has elements:
         add a[0] to the end of c
         remove a[0] from a
     while b has elements:
         add b[0] to the end of c
         remove b[0] from b
     return c
end func
```

Example: Merging two sorted lists

1 3 7 9 | 2 4 6 8

1. Make a new empty list Q = [ ]
2. Compare the heads of the lists
3. Pop the smaller to the Q
4. Go back to 2 until one list is empty
5. Append the remaining elements into Q.

Hint: you can use the .pop() method of Python list

1st round: we compare 1 and 2. Since 1 < 2, we remove 1 from the left list and put it into Q. So, the left list (in red) becomes [3, 7, 9]; the right doesn't change; and Q = [1].

2nd round: we compare 3 and 2. Since 2<3, we remove 2 from the right list and put it into Q. So, the right list becomes [4, 6, 8], the left doesn't change; and Q = [1, 2].

……

After 4 rounds, the left list will have one element [9], and the right will be empty. So, we will append the remaining element to Q.

Merge Sort: Pseudocode

```
func merge_sort(var a as array):
    if n == 1:   #n is the length of a
        return a
    var l1 as array = [a[0],...,a[n/2]]
    var l2 as array = [a[n/2+1],...,a[n]]
    l1 = merge_sort(l1)
    l2 = merge_sort(l2)
    return merge(l1, l2)
end func
```

```
func merge(var a as array, var b as array):
    var c as array
    while a and b have elements:
        if a[0] > b[0]:
            add b[0] to the end of c
            remove b[0] from b
        else:
            add a[0] to the end of c
            Remove a[0] from a
    while a has elements:
        add a[0] to the end of c
        remove a[0] from a
    while b has elements:
        add b[0] to the end of c
        remove b[0] from b
    return c
end func
```

[Exercise]

Translate pseudocode of merge() into a function!

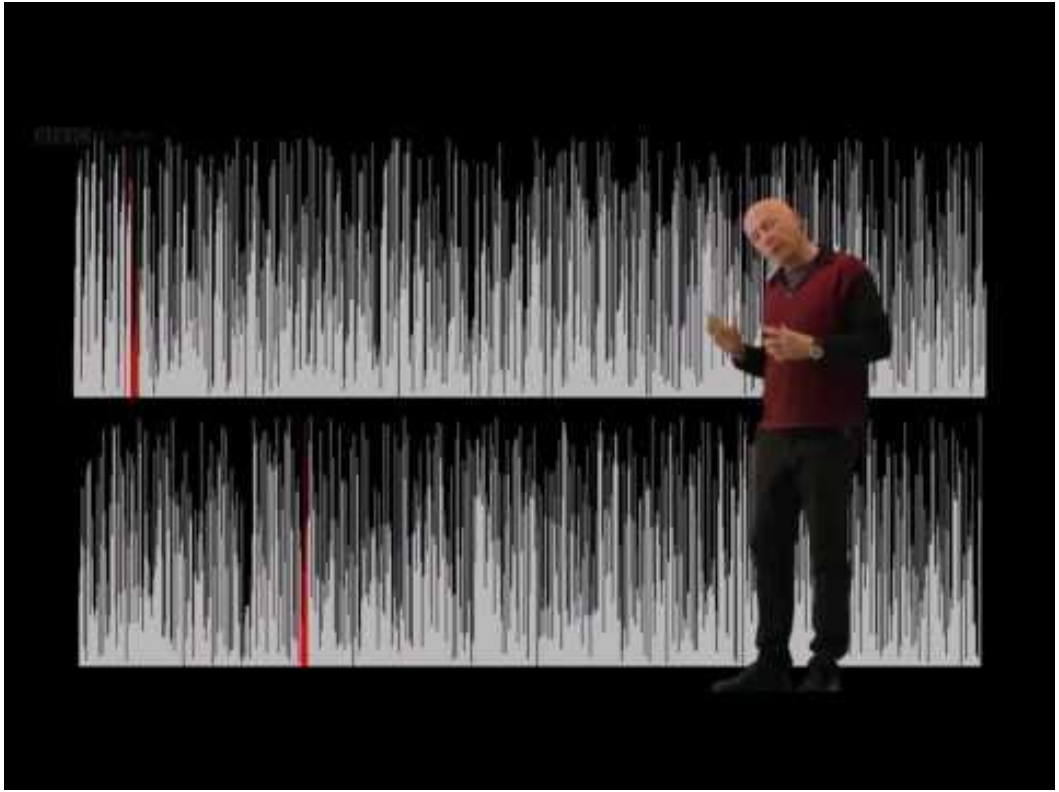
Analysis: Merge vs. Bubble

- Which algorithm is better?
  - What does “better” mean?

- Some algorithms are “better” than others.
  - Usability
  - Simplicity / Readability
  - **Time**
  - **Space**
  - ‘Beauty’/elegance

Usually, we evaluate the performance of an algorithm on its time consumption and space consumption.

Analysis: Merge vs. Bubble



Analysis: Merge vs. Bubble

- In general, merge sort is far better than bubble sort

- Exceptions:
  - The list is sorted!
  - The memory space is limited.

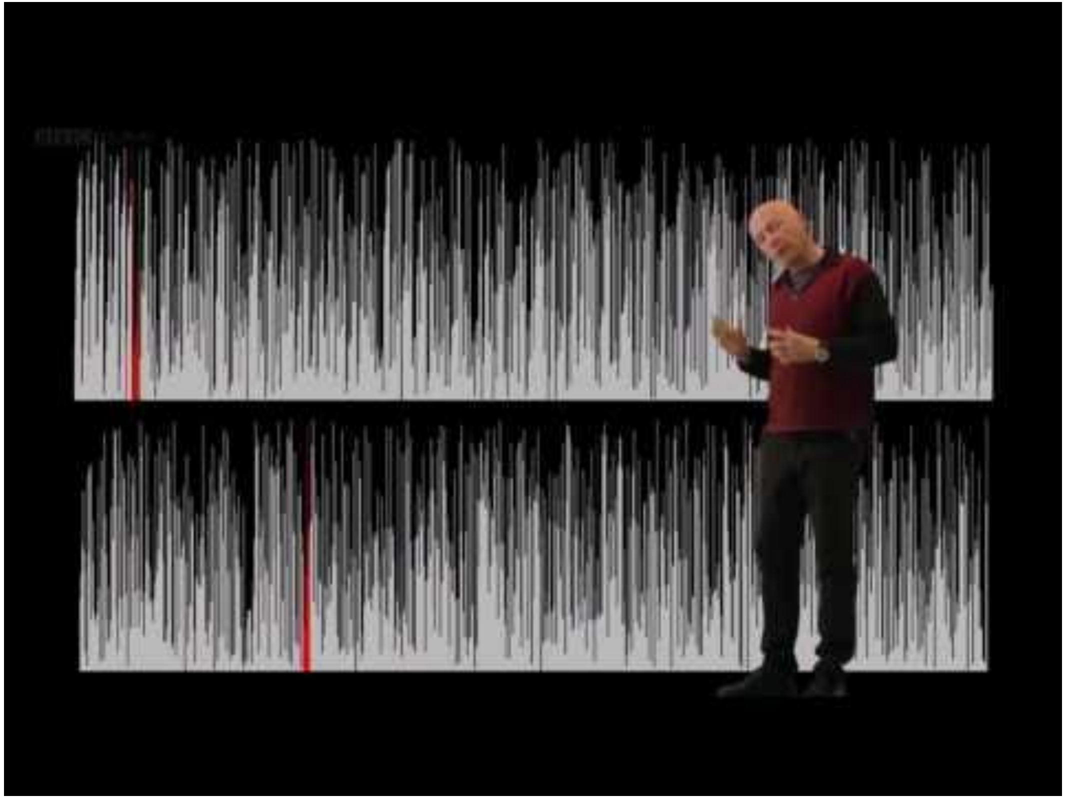
Appendix: sorting in different cases

https://www.toptal.com/developers/sorting-algorithms

Appendix: other sorting algorithms

- Bucket sort
  - Divide values into buckets, sort buckets then concatenate them
- Insertion sort
  - Build the sorted list by inserting one element at a time
- Heap sort
  - Uses trees (a type of data structure)
- Quick sort
  - Based upon pivots

Appendix: Travelling Salesman Problem (TSP)



BBC Documentary

- Algorithms - The Secret Rules of Modern Living

    - The research of algorithm is a very important branch of computer science.
    - **[READING]** “9 Algorithms that changed the future” by John MacCormick- a set of algorithms you use every day, including:
        - Page rank – web search (typical case: Google search engine)
        - Encryption
        - Error correction