Principles of Computer Science II

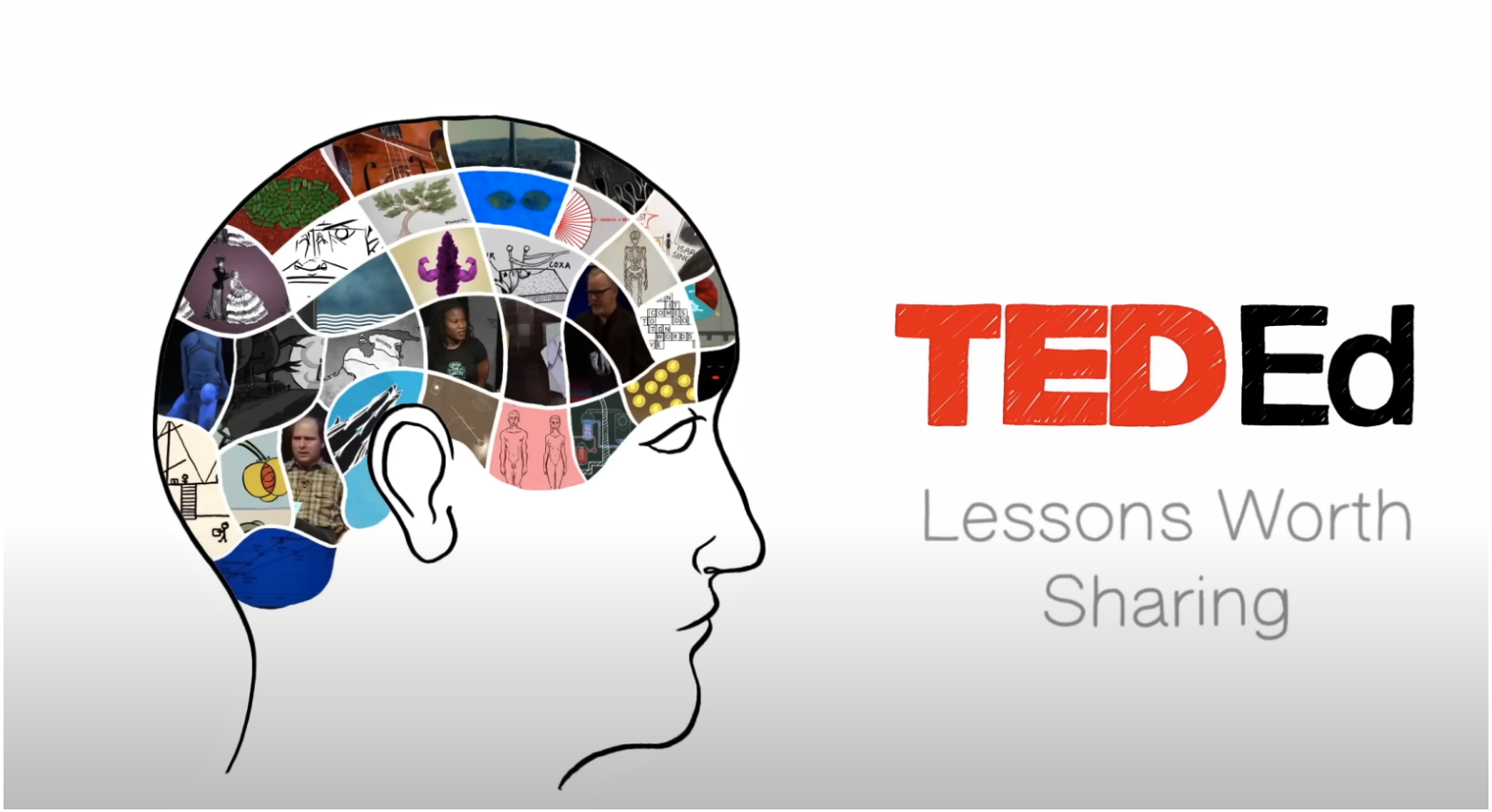
Sorting Algorithms

Marco Zecchini

Sapienza University of Rome

Lecture 3

Introductory Video

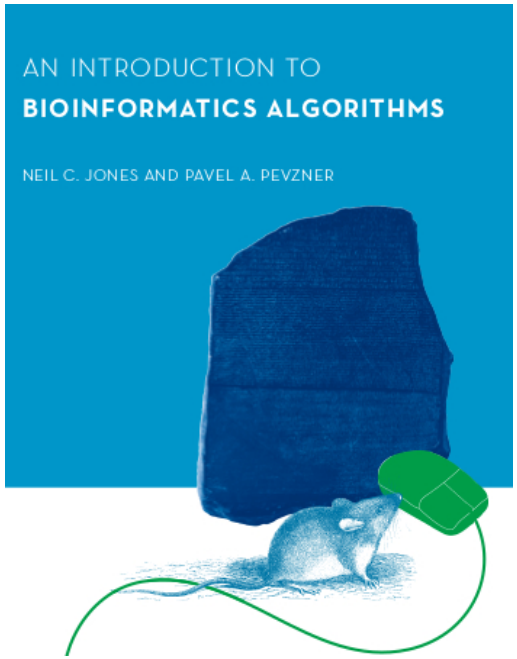


https://www.youtube.com/watch?v=WaNLJf8xzC4

Sorting Problem

Jones, Pevzner: An Introduction to Bioinformatics Algorithms. MIT Press, 2004

Section 2.6 - Sorting Problem



Sorting Problem:
Sort a list of integers.

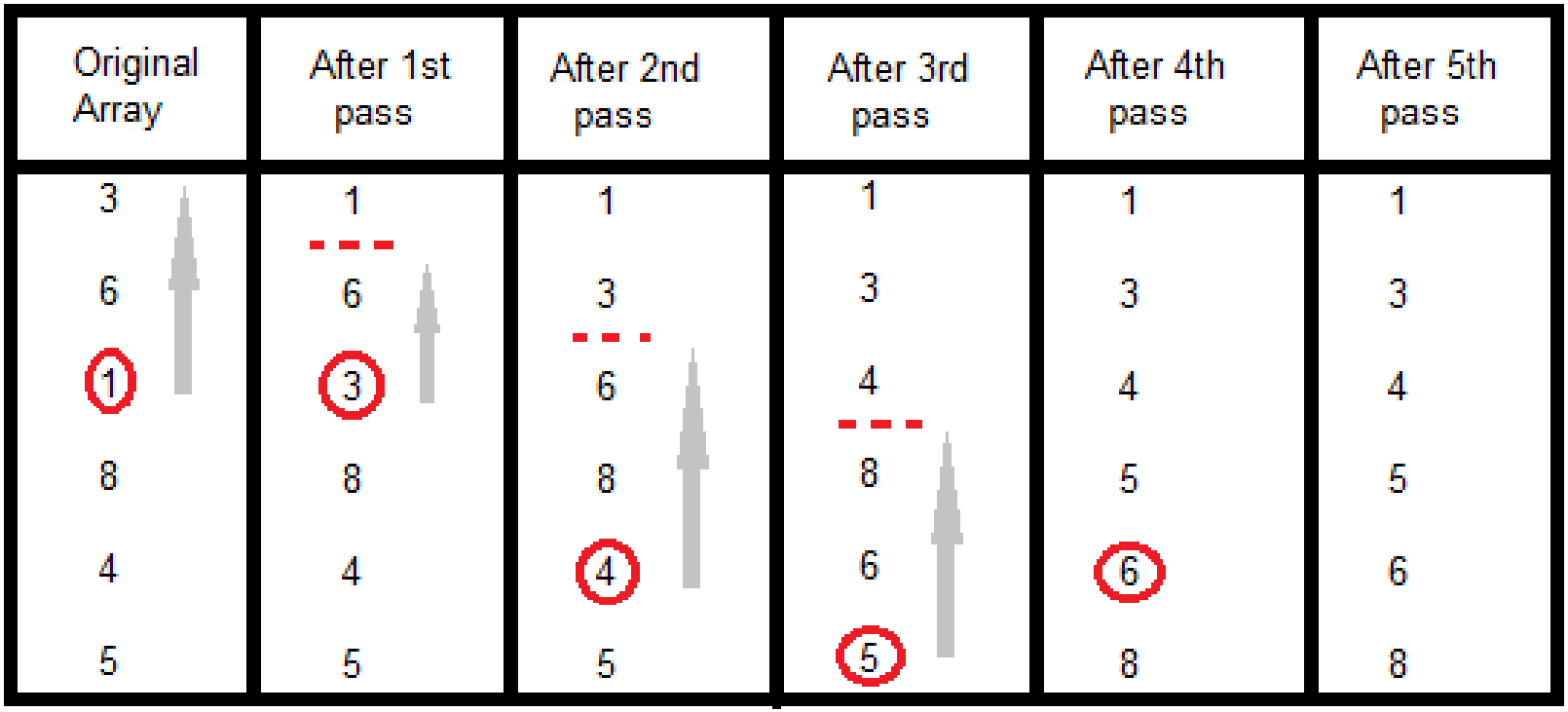
Input: A list of n distinct integers $\mathbf{a} = (a_1, a_2, \ldots, a_n)$.

Output: Sorted list of integers, that is, a reordering $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ of integers from $\mathbf{a}$ such that $b_1 < b_2 < \cdots < b_n$.

Selection Sort Algorithm

This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

Selection Sort: Example



Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

Selection Sort Code

```
a = [5, 1, 6, 2, 4, 3]
for i in range(0, len(a)):
    min = i
    # find the smallest element in the rest of the array
    for j in range(i + 1, len(a) - 1):
        if a[j] < a[min]:
            min = j

    # swap the elements
    temp = a[j]
    a[j] = a[min]
    a[min] = temp
```

How good is Selection Sort?

- How many swaps are required until the list is sorted? (in the worst case scenario)

How good is Selection Sort?

- How many swaps are required until the list is sorted? (in the worst case scenario)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...

How good is Selection Sort?

- How many swaps are required until the list is sorted? (in the worst case scenario)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ... +3+2+1 swaps are required

How good is Selection Sort?

- How many swaps are required until the list is sorted? (in the worst case scenario)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - **time complexity is** $O(n^2)$

How good is Selection Sort?

- How many swaps are required until the list is sorted? (in the worst case scenario)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - **time complexity is** $O(n^2)$
- How much memory is needed ?

How good is Selection Sort?

- How many swaps are required until the list is sorted? (in the worst case scenario)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - **time complexity is** $O(n^2)$
- How much memory is needed ?
  - 2 additional slot (min and temp) - constant, $O(1)$!

Bubble Sort Algorithm

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory. Bubble Sort compares all the elements one by one and sort them according to to their values.

- It is called Bubble sort, because with each iteration the *largest* element in the list bubbles up towards the last place, just like a water bubble rises up to the water surface.
- Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Bubble Sorting: Example

Lets take this Array.

5 1 6 2 4 3

1 5 6 2 4 3

1 5 2 6 4 3

1 5 2 4 6 3

1 5 2 4 3 6

Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

Bubble Sort Code

```
a = [5, 1, 6, 2, 4, 3]
for i in range(0, len(a)):
    for j in range(0, len(a) - i - 1):
        if a[j] > a[j+1]:
            temp = a[j]
            a[j] = a[j+1]
            a[j+1] = temp
```

Bubble Sort Code

```
a = [5, 1, 6, 2, 4, 3]
for i in range(0, len(a)):
    for j in range(0, len(a) - i - 1):
        if a[j] > a[j+1]:
            temp = a[j]
            a[j] = a[j+1]
            a[j+1] = temp
```

- After the `1st` pass (`i = 0`), the largest element is correctly placed at the **last position**.
- After the `2nd` pass (`i = 1`), the last **two** elements are already sorted.

Therefore, at each iteration we can shorten the range of the inner loop: $j \in [0, \texttt{len(a)} - i - 1)$ so that we do not recompare elements that are already in their final positions.

Bubble Sort Code

```
a = [5, 1, 6, 2, 4, 3]
for i in range(0, len(a)):
    for j in range(0, len(a) - i - 1):
        if a[j] > a[j+1]:
            temp = a[j]
            a[j] = a[j+1]
            a[j+1] = temp
```

- The above algorithm is not efficient because as per the above logic, the for-loop will keep executing for six iterations even if the list gets sorted after the second iteration.

Bubble Sort Code: Version 2

- We can insert a flag and can keep checking whether swapping of elements is taking place or not in the following iteration.
- If no swapping is taking place, it means the list is sorted and we can jump out of the for loop, instead executing all the iterations.

```
a = [5, 1, 6, 2, 4, 3]
for i in range(0, len(a)):
    swapped = False  # flag to track if a swap occurs
    for j in range(0, len(a) - i - 1):
        if a[j] > a[j + 1]:
            temp = a[j]
            a[j] = a[j + 1]
            a[j + 1] = temp
            swapped = True  # mark that a swap happened
    if not swapped:
        break  # exit early, list is already sorted
```

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - time complexity $O(n^2)$

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - time complexity $O(n^2)$
- Is there a "best" case ?

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - time complexity $O(n^2)$
- Is there a "best" case ?
  - The list is already sorted

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - time complexity $O(n^2)$
- Is there a "best" case ?
  - The list is already sorted
  - How many loops are required in the optimized version?

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - time complexity $O(n^2)$
- Is there a "best" case ?
  - The list is already sorted
  - How many loops are required in the optimized version?
  - N swaps are required - time complexity $O(n)$

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - time complexity $O(n^2)$
- Is there a "best" case ?
  - The list is already sorted
  - How many loops are required in the optimized version?
  - N swaps are required - time complexity $O(n)$
- How much memory is needed ?

How good is Bubble Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required - time complexity $O(n^2)$
- Is there a "best" case ?
  - The list is already sorted
  - How many loops are required in the optimized version?
  - N swaps are required - time complexity $O(n)$
- How much memory is needed ?
  - 1 additional slot (naive solution)
  - 2 additional slot (temp + flag in the optimized solution)
  - ...however, constant!

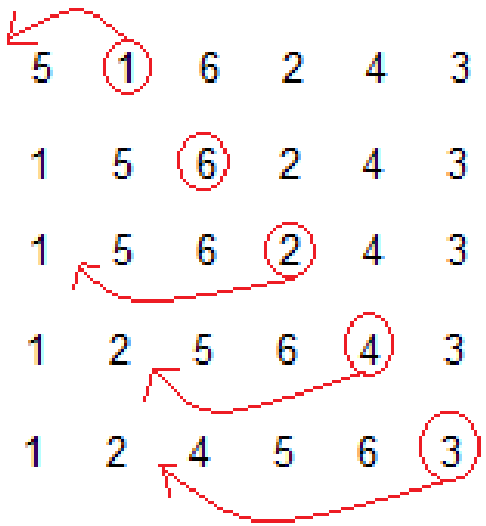
Insert Sort Algorithm

A simple Sorting algorithm which sorts the list by shifting elements one by one.

- It has one of the simplest implementation
- It is efficient for smaller data sets, but very inefficient for larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- It is better than Selection Sort and Bubble Sort algorithms.
- Like Bubble Sorting, insertion sort also requires a single additional memory space.

Insertion Sort: Example

Lets take this Array.



(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

Insertion Sort Code

```
a = [5, 1, 6, 2, 4, 3]
for i in range(1, len(a)):
    key = a[i] # in the first iteration, the 2nd elem is the
         key
    j = i - 1
    while j >= 0 and key < a[j]:
        a[j+1] = a[j] # it ''brings'' a[j] back in the list
        j -= 1
    a[j+1] = key # from a[0] to the a[i] now the sublist is
        sorted
```

- key: we put each element of the list, in each pass, starting from the second element: a[1].
- using the while loop, we iterate, until j becomes equal to zero or we find an element which is greater than key, and then we insert the key at that position.

How good is Insertion Sort?

- How many swaps are required until the list is sorted? (in the worst case)

How good is Insertion Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ... +3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required

How good is Insertion Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ... +3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- Is there a "best" case ?

How good is Insertion Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- Is there a "best" case ?
  - The list is already sorted
  - N swaps are required... $key < a[j]$ condition of the while is never met

How good is Insertion Sort?

- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ... +3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- Is there a "best" case ?
  - The list is already sorted
  - N swaps are required... $key < a[j]$ condition of the while is never met
- How much memory is needed ?

How good is Insertion Sort?

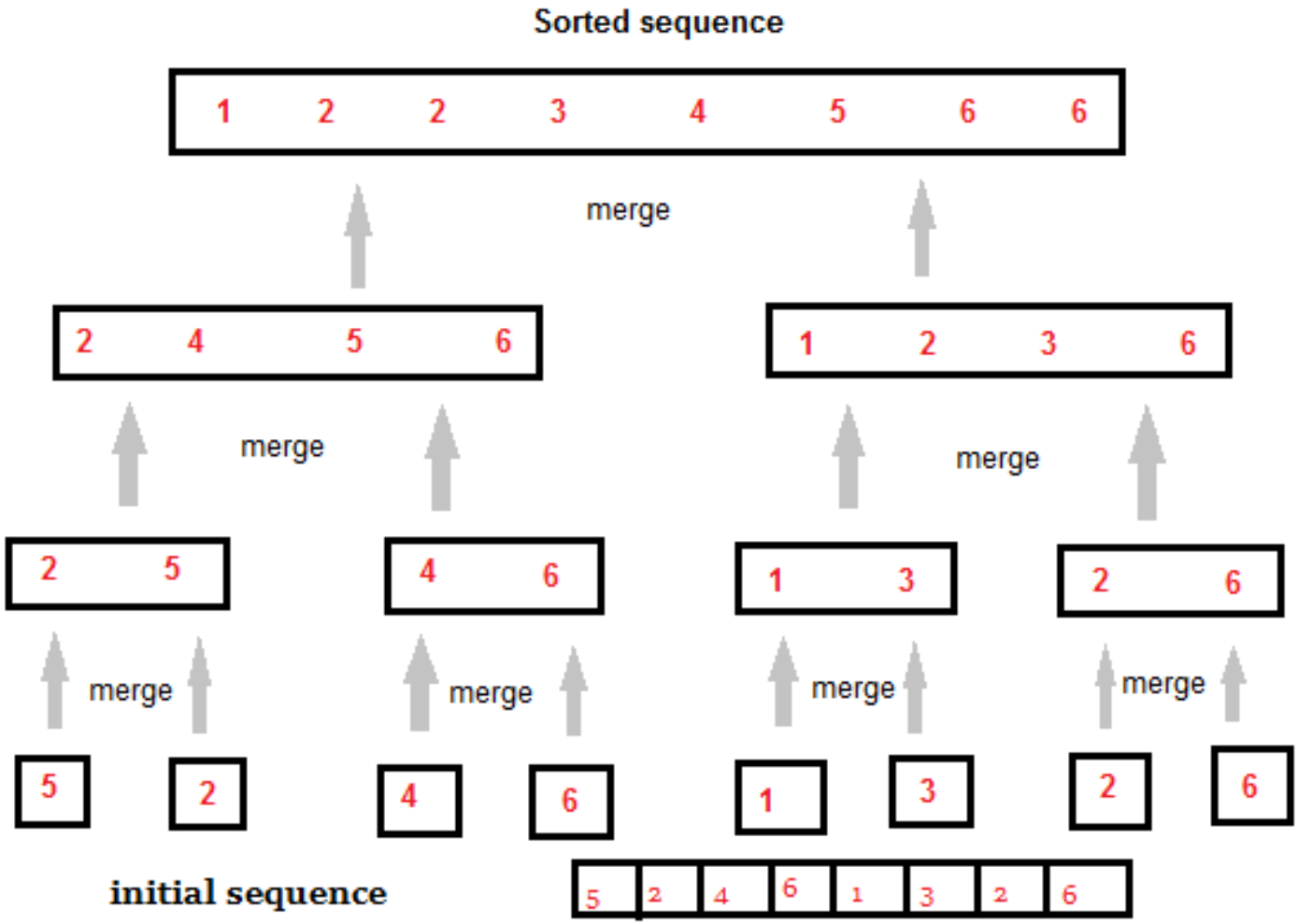
- How many swaps are required until the list is sorted? (in the worst case)
  - 1^{st} loop: n - 1
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ... +3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- Is there a "best" case ?
  - The list is already sorted
  - N swaps are required... $key < a[j]$ condition of the while is never met
- How much memory is needed ?
  - 2 additional slot (key, j)
  - constant!

Merge Sort Algorithm

In Merge Sort the unsorted list is divided into N sublists, each having one element, because a list consisting of one element is always sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and in the end, only one sorted list is produced.

- Divide and Conquer algorithm
- Performance always same for Worst, Average, Best case

Merge Sort: Example



Merge Sort Code

```
a = [25, 52, 37, 63, 14, 17, 8, 6]

def mergesort(list):
    if len(list) == 1:
        return list

    left = list[0: len(list) // 2]
    right = list[len(list) // 2:]

    left = mergesort(left)
    right = mergesort(right)

    return merge(left, right)
```

Merge Sort Code

```
def merge(left, right):
    result = []
    # we remove one element either from left or right
    while len(left) > 0 and len(right) > 0:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    # if right is empty
    while len(left) > 0:
        result.append(left.pop(0))
    # if left is empty
    while len(right) > 0:
        result.append(right.pop(0))

    return result

print("Before: ", a)
r = mergesort(a)
print("After: ", r)
```

Merge Sort - Iterative version

```
def iterative_mergesort(a):
    size = 1
    n = len(a)
    while size < n:
        for start in range(0, n, 2 * size):
            mid = start + size
            end = min(start + 2 * size, n)
            left = a[start:mid]
            right = a[mid:end]
            a[start:end] = merge(left, right)
        size *= 2
    return a
```

How good is Merge Sort?

- How many swaps are required until the list is sorted?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n swaps
  - In total $n \log n$ swaps
- How much memory is needed ?
  - A temporary array `result` of size $|\texttt{left} + \texttt{right}|$ is needed for merging.
  - In the last recursive step, `result` is of size $n/2 + n/2 = n$ - space complexity $O(n)$ (not constant).
  - Optimized version works in-place

Quick Sort Algorithm

Quick Sort is very fast and requires very less additional space. It is based on the rule of Divide and Conquer. This algorithm divides the list into three main parts :

- Elements less than the Pivot element
- Pivot element(Central element)
- Elements greater than the pivot element

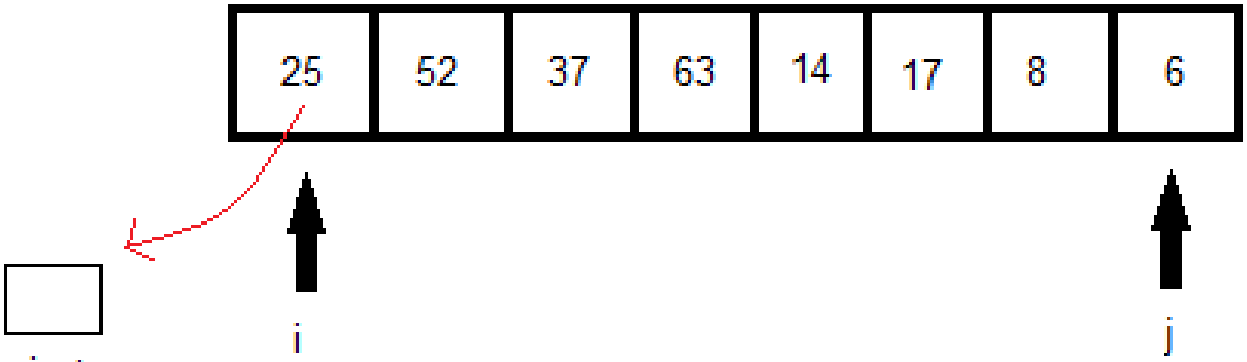
- Sorts any list very quickly
- Performance depends on the selection of the Pivot element

Quick Sort: Example

List: 25 52 37 63 14 17 8 6

- We pick 25 as the pivot.
- All the elements smaller to it on its left,
- All the elements larger than to its right.
- After the first pass the list looks like:
  6 8 17 14 25 63 37 52
- Now we sort two separate lists:
  6 8 17 14 and 63 37 52
- We apply the same logic, and we keep doing this until the complete list is sorted.

Quick Sort: Example



if both sides we find the element not satisfying their respective conditions, we swap them. And keep repeating this.

DIVIDE AND CONQUER - QUICK SORT

Quick Sort Code

```
def quicksort(list, p, r):
    # p = Start index of the subarray
    # r = End index of the subarray
    if (p < r):
        q = partition(list, p, r)# Partition index (pivot
            position after rearranging)
        quicksort(list, p, q);
        quicksort(list, q + 1, r);

print("Before: ", a)
quicksort(a, 0, len(a) - 1)
print("After: ", a)
```

Quick Sort Code

```
a = [25, 52, 37, 63, 14, 17, 8, 6]

def partition(list, p, r):
    pivot = list[p] # select a pivot, the value might be the
         median
    i, j = p, r
    while(1):
        # find the first element greater than the pivot
        while(list[i] < pivot and list[i] != pivot):
            i += 1
        # find the first element smaller than the pivot
        while(list[j] > pivot and list[j] != pivot):
            j -= 1

        if(i < j): # swap them
            temp = list[i]
            list[i] = list[j]
            list[j] = temp
        else:
            return j
```

How good is Quick Sort?

- How many swaps are required until the list is sorted?

How good is Quick Sort?

- How many swaps are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?

How good is Quick Sort?

- How many swaps are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1 (bring all the element in one of the two partition)
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required

How good is Quick Sort?

- How many swaps are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1 (bring all the element in one of the two partition)
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- What if we choose the median item as pivot?

How good is Quick Sort?

- How many swaps are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1 (bring all the element in one of the two partition)
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- What if we choose the median item as pivot?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n swaps
  - In total $n \log n$ swaps

How good is Quick Sort?

- How many swaps are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1 (bring all the element in one of the two partition)
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- What if we choose the median item as pivot?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n swaps
  - In total $n \log n$ swaps
- How much memory is needed ?

How good is Quick Sort?

- How many swaps are required until the list is sorted?
- What if we choose the smallest or the largest item as pivot?
  - 1^{st} loop: n - 1 (bring all the element in one of the two partition)
  - 2^{nd} loop: n - 2
  - ...
  - (n-1)+(n-2)+(n-3)+ ...+3+2+1 swaps are required
  - $\sum \frac{n(n-1)}{2}$ swaps are required
- What if we choose the median item as pivot?
  - 1^{st} loop: two lists $\frac{n}{2}$ each
  - 2^{nd} loop: four lists $\frac{n}{4}$ each
  - ...
  - $\log n$ steps
  - For each partition we do n swaps
  - In total $n \log n$ swaps
- How much memory is needed ?
  - 2 small additional slots.

Algorithm Design Techniques

For sorting, we solved the same problem using different techniques (solve each subproblem separately, divide-and-conquer, randomization).

Algorithm Design Techniques

Computer scientists have discovered that many algorithms share similar ideas, even though they solve very different problems.

Daily life problem

Jones, Pevzner: An Introduction to Bioinformatics Algorithms.
MIT Press, 2004
Section 2.9

Exhaustive Search

- You ignore that the phone is ringing
- You walk through every possible angle of the room to find the phone
- You eventually find the phone, but you won't be able to answer

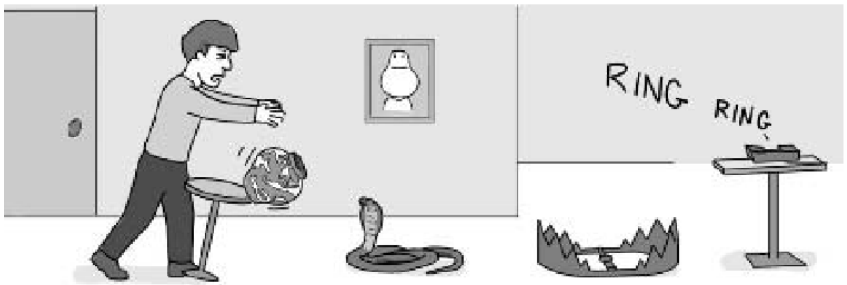
Exhaustive Search

- You ignore that the phone is ringing
- You walk through every possible angle of the room to find the phone
- You eventually find the phone, but you won't be able to answer
- You can optimize such an approach by *omitting or pruning* part the alternatives (e.g., if the phone is ringing above your head, just look everywhere upstairs!)

Exhaustive Search

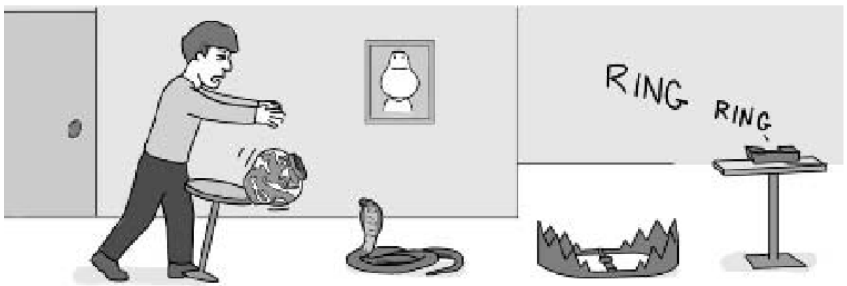
- Is one of the algorithms for sorting doing brute force?
- No, how would that be?

Greedy Algorithms



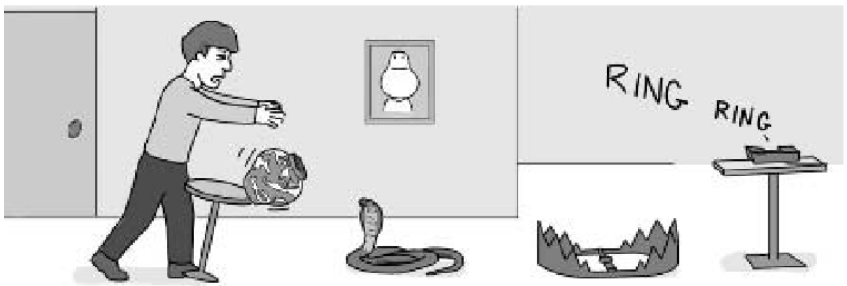
- Walk in the direction of the telephone's ringing until you found it.
- If there is a wall (or an expensive and fragile vase) between you and the phone, prevents you from finding the phone.
- Unfortunately, these sorts of difficulties frequently occur in most realistic problems.

Greedy Algorithms



- Is one of the algorithms for sorting greedy?

Greedy Algorithms



- Is one of the algorithms for sorting greedy?
- Selection sort

Dynamic Programming problems

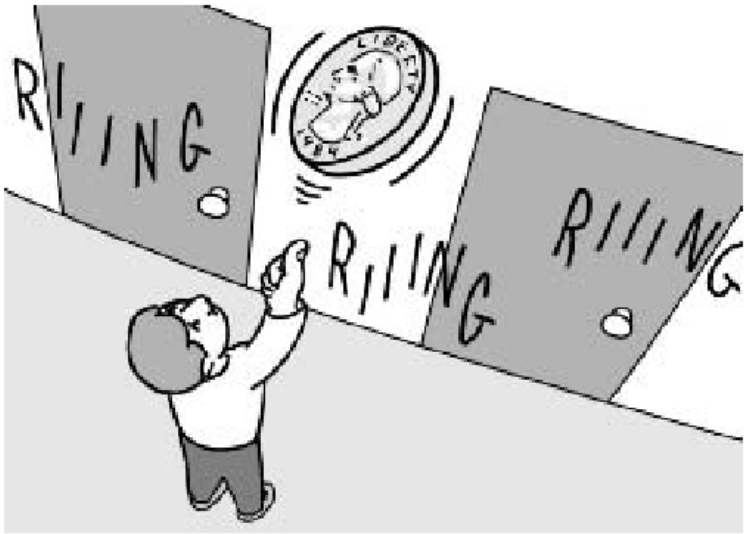
	0	1	2	3	4	5	6	7	8	9	10
0	*	←	*	←	*	←	*	←	*	←	*
1	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
2	*	←	*	←	*	←	*	←	*	←	*
3	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
4	*	←	*	←	*	←	*	←	*	←	*
5	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
6	*	←	*	←	*	←	*	←	*	←	*
7	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
8	*	←	*	←	*	←	*	←	*	←	*
9	↑	↖	↑	↖	↑	↖	↑	↖	↑	↖	↑
10	*	←	*	←	*	←	*	←	*	←	*

- Split a problem into subproblems and solve each subproblem to solve the general problem
- Not applicable to the problem of the phone.
- You want to freshen all the rooms of your house.
- You solve the problem room by room by turning on the AC at the proper temperature (depending on sun exposition, for example) in each room.

Divide and conquer Algorithms

- Split the problem of finding the phone into subproblems (e.g., look in different rooms) and solve each subproblem to solve the general problem and then *merge* the solutions
- We have seen two algorithms working in this way: Merge Sort and Quick Sort
- This approach goes along with recursion

Randomized Algorithms



- Toss a coin to decide whether you want to start your search of the phone on the first floor if the coin comes up heads or on the second floor if the coin comes up tails (you can also use a die).
- We have seen an algorithm working this way: Quick Sort

Machine Learning Approach

- Instead of solving a problem from scratch, a system can **learn from experience**.
- Machine learning uses **past data** to predict or guide future actions.
- Example: if the phone is usually found

  $$P(\text{bathroom}) = 0.8,\ P(\text{bedroom}) = 0.15,\ P(\text{kitchen}) = 0.05,$$

  then start searching where it's most likely to be.
- The algorithm becomes **adaptive**, improving with each observation.