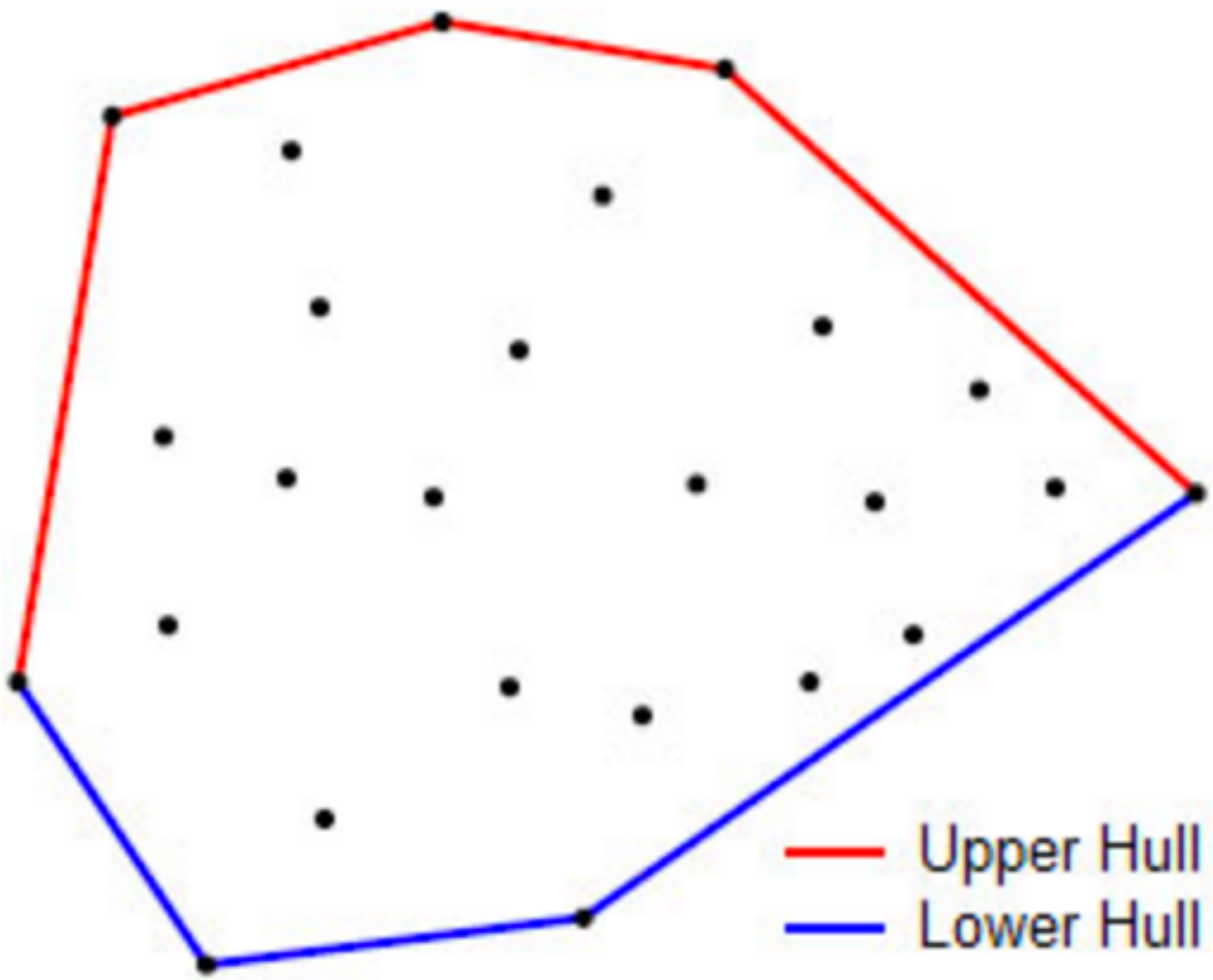

2.2 Graham's scan

The Graham's Algorithm first explicitly sorts the points in O(nlogn) and then applies a linear-time scanning algorithm to finish building the hull.

http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_cs_07w/Webprojects/Zhaobo_hull/

Sorting Lists

Input:

- A list (not necessarily sorted) of ‘orderable’ element types
- For example, in Python:
  - `the_list = [5,1.5,3,-4.0]` is fine
  - `the_list = [1,'hj',0,'j']` is not
    - Unless you define your own comparison function

Output:

- A list with the same elements as the input list BUT sorted in increasing order.

Algorithm	Best case Time complexity	Worst case Time complexity	Stability
Bubble sort			
Selection sort			
Insertion sort	O(n)	$O(n^2)$	

Bubble Sort

Main idea:
Lighter bubbles rise to the top,
Heavier ones sink to the bottom.

smaller elements "bubble" to the front of the list, larger sink to the end.

Bubble Sort: Python Code

Algorithm BubbleSort(L)
// Sorts a list using bubble sort
// Input: A list of orderable items
// Output: A list sorted in increasing order

```
n ⟵ length(L)
i ⟵ 0
while i< n-1 {
    j ⟵ 0
    while j< n-1 {
        if L[j] > L[j+1] {
            swap L[j] and L[j+1]
        }
        j ⟵ j + 1
    }
    i ⟵ i + 1
}
```

```python
def bubble_sort(the_list):
    n = len(the_list)
    i = 0
    while i < n - 1:
        j = 0
        while j < n - 1:
            if the_list[j] > the_list[j + 1]:
                swap(the_list, j, j + 1)
            j += 1
        i += 1
```

```python
def swap(the_list, i, j):
    tmp = the_list[i]
    the_list[i] = the_list[j]
    the_list[j] = tmp
```

```python
def bubble_sort(the_list):
    n = len(the_list)
    for i in range(n - 1):
        for j in range(n - 1):
            if the_list[j] > the_list[j + 1]:
                swap(the_list, j, j + 1)


def swap(the_list, i, j):
    the_list[i], the_list[j] = the_list[j], the_list[i]
```

```
   def bubble_sort(the_list):
t1     n = len(the_list)
t2     for i in range(n - 1):
t3         for j in range(n - 1):
t4             if the_list[j] > the_list[j + 1]:
t5                 swap(the_list, j, j + 1)
```

best case: [1, 2, 3, 4, 5, …, n]

no swaps

```
    def bubble_sort(the_list):
t1      n = len(the_list)
t2      for i in range(n - 1):
t3          for j in range(n - 1):
t4              if the_list[j] > the_list[j + 1]:
t5                  swap(the_list, j, j + 1)
```

$\mathbf{c} = t_2$

$\mathbf{d} = t_3 + t_4$

$$\underbrace{\mathbf{c} + (n-1)\mathbf{d}}_{i=0} + \underbrace{\mathbf{c} + (n-1)\mathbf{d}}_{i=1} + \underbrace{\mathbf{c} + (n-1)\mathbf{d}}_{i=2} + \dots + \underbrace{\mathbf{c} + (n-1)\mathbf{d}}_{i=n-2}$$

$$(n-1)[\mathbf{c} + (n-1)\mathbf{d}]$$

$$n^2\mathbf{d} + n(\mathbf{c} - 2\mathbf{d}) + (\mathbf{d} - \mathbf{c})$$

$$O(n^2)$$

worst case: [n, n-1,n-2, …, 2, 1]

every swap

```
    def bubble_sort(the_list):
t1      n = len(the_list)
t2      for i in range(n - 1):
t3          for j in range(n - 1):
t4              if the_list[j] > the_list[j + 1]:
t5                  swap(the_list, j, j + 1)
```

$c = t_2$

$k = t_3 + t_4 + t_5$

$$\underset{i=0}{c + (n-1)k} + \underset{i=1}{c + (n-1)k} + \underset{i=2}{c + (n-1)k} + \ldots + \underset{i=n-2}{c + (n-1)k}$$

$$(n-1)[c + (n-1)k]$$

$$n^2k + n(c-2k)+(k-c)$$

$O(n^2)$

Bubble Sort

$n^2\mathbf{d} + n(\mathbf{c}-2\mathbf{d})+(\mathbf{d}-\mathbf{c})$ $\qquad$ $\mathbf{O(n^2)}$

$n^2\mathbf{k} + n(\mathbf{c}-2\mathbf{k})+(\mathbf{k}-\mathbf{c})$ $\qquad$ $\mathbf{O(n^2)}$

k>d, but in Big O constants do not matter

Bubble Sort: Time complexity

```
def bubble_sort(the_list):
    n = len(the_list)
    for i in range(n - 1):
        for j in range(n - 1):
            if the_list[j] > the_list[j + 1]:
                swap(the_list, j, j + 1)
```

n times

n times

We cannot stop any of the two loops early.

$O(n^2)$ = worst case = best case

Alternative version

```python
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n - 1, 0, -1):
        for i in range(mark):
            if the_list[i] > the_list[i + 1]:
                swap(the_list, i, i + 1)
```

mark times

n times

Intuition: nested loops, both dependent on n, every operation on inner loop performed a fixed number of times: the worst case is going to be O(n^2)

```python
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n - 1, 0, -1):
        swapped = False
        for i in range(mark):
            if the_list[i] > the_list[i + 1]:
                swap(the_list, i, i + 1)
                swapped = True
        if not swapped:
            break
```

Stop as soon as the list is sorted.

Improved bubble sort

```python
def bubble_sort(the_list):
    n = len(the_list)
    for mark in range(n - 1, 0, -1):
        swapped = False
        for i in range(mark):
            if the_list[i] > the_list[i + 1]:
                swap(the_list, i, i + 1)
                swapped = True
        if not swapped:
            break
```

- Can you leave any of the two loops early?
- Best case $\neq$ Worst case
- **Best case** is a sorted list: **O(n)**
- **Worst case** is list in reverse order: $\mathbf{O(n^2)}$

Selection Sort

(find minimum,
put it where it belongs,
reduce)

Selection Sort: Code

Algorithm SelectionSort(L)

// Sorts a list using selection sort

// Input: A list of orderable items

// Output: A list sorted in increasing order

n ⟵ length(L)

k ⟵ 0

while k < n {

  Find the minimum item in L[k:n-1] {

    Put the item in the correct position

  }

  k ⟵ k + 1

}

```python
def selection_sort(the_list):
    n = len(the_list)
    for k in range(n):
        min_position = find_minimum(the_list, k)
        swap(the_list, k, min_position)
```

```python
def find_minimum(the_list, starting_index):
    min_position = starting_index
    n = len(the_list)
    for i in range(starting_index, n):
        if the_list[i] < the_list[min_position]:
            min_position = i
    return min_position
```

Selection Sort: Code

n-1 times

```
def selection_sort(the_list):
    n = len(the_list)
    for mark in range(n - 1):
        min_index = find_minimum(the_list, mark)
        swap(the_list, mark, min_index)
```

n-(mark+1) times each

fixed

```
def find_minimum(the_list, mark):
    position_minimum = mark
    n = len(the_list)
    for i in range(mark + 1, n):
        if the_list[i] < the_list[position_minimum]:
            position_minimum = i
    return position_minimum
```

Can we stop any of the two loops early?

Algorithm	Best case Time complexity	Worst case Time complexity	Stability
Bubble sort	$O(n)$	$O(n^2)$	
Selection sort	$O(n^2)$	$O(n^2)$	
Insertion sort	$O(n)$	$O(n^2)$	

Stability

Stable sorting

A sorting algorithm is stable if it **maintains the relative order among elements.**

8	3	8	6	3
a	b	c	d	e

3	3	6	8	8
b	e	d	a	c

The **relative order** is preserved
(b before e, a before c)

3	3	6	8	8
e	b	d	a	c

The **relative order**
may not be preserved

Name	Mark
Ann	100
Brendon	90
Cheng	100
Daniel	50

Name	Mark
Daniel	50
Brendon	90
Cheng	100
Ann	100

Cheng before Ann

Name	Mark
Daniel	50
Brendon	90
Ann	100
Cheng	100

Ann before Cheng

stable:

the relative order of elements with the same value is maintained.

Algorithm	Best case Time complexity	Worst case Time complexity	Stability
Bubble sort	$O(n)$	$O(n^2)$	**Yes**
Selection sort	$O(n^2)$	$O(n^2)$	**No**
Insertion sort	$O(n)$	$O(n^2)$	**Yes**

Summary

You need to understand and be able to implement the following simple sorting algorithms knowing their time complexity and stability properties:

- Bubble sort
- Selection sort
- Insertion sort