Sorting Algorithms

Julien BESTARD

Contents

1 Some information before the algorithms **3**
1.1 Link to PDF . . . 3
1.2 In-place/Not in-place function . . . 3
1.3 Function and Procedure (Reminder) . . . 3

2 Selection Sort **4**
2.0 Explanation . . . 4
2.1 Algorithm . . . 4
2.21 Complexity . . . 4

3 Insert Sort **5**
3.0 Explanation . . . 5
3.1 In Place . . . 5
3.1.1 Algorithm . . . 5
3.1.2 Complexity . . . 5
3.2 Not In place . . . 6
3.1.1 Algorithm . . . 6
3.2.2 Complexity . . . 6
3.3 In place using binary search . . . 7
3.3.2 Complexity . . . 7
3.3.1 Algorithm . . . 7

4 Bubble Sort **8**
4.1 Bubble Sort . . . 8
4.1.0 Explanation . . . 8
4.1.1 Complexity . . . 8
4.1.2 Iterative algorithm . . . 8
4.2 Shaker Sort . . . 9
4.2.0 Explanation . . . 9
4.2.1 Complexity . . . 9
4.2.2 Iterative algorithm . . . 9

5 Divide and Conquer **10**
5.1 Merge Sort . . . 10
5.1.0 Explanation . . . 10
5.1.1 Complexity . . . 10
5.1.2 Algorithm . . . 10
5.2 Quick Sort . . . 12
5.2.1 Complexity . . . 12
5.2.2 Algorithm . . . 12

6 Complexity Recap **13**

1 Some information before the algorithms

In-place/Not in-place function

Link to the PDF :

> **French doc :** https://moodle.cri.epita.fr/pluginfile.php/68426/mod_label/intro/S2_td0_sorts_fr.pdf
>
> **English doc :** https://moodle.cri.epita.fr/pluginfile.php/68426/mod_label/intro/S2_td0_sorts_en.pdf

In-place/Not in-place function

An in-place function is a function that directly modifies the variable in parameter. It does not need to return anything.

A function not in place modifies the variables in parameters. It, unlike the in-place function, returns something.

Function and Procedure (Reminder)

Function (not in-place) :

- Return a result
- No edge effect

procedure (in-place) :

- Modifies the environment (= edge effect)
- No return

2 Selection Sort

Explanation

The principle of sorting by selection is to go and find the smallest element of the list to put it in first, then to start again from the second element and to go and find the smallest element of the list to put it in second, etc...

Algorithm

```
def select_sort(L:list) -> None:
    """
    Sort a list using the select sort algorithm

    Parameters
    ----------
    L : List
        unsorted list.

    Returns
    -------
    Nothing.
    """
    length = len(L)
    for i in range(length):
        index = i
            for j in range(i+1, length):
                if L[j] < L[index]:
                    index = j
                if i != index:
                    L[index], L[i] = L[i], L[index]
```

Complexity

Number of comparisons : at worst / at best
Number of copies of elements : at worst / at best

3 Insert Sort

Explanation

This is the card player sort. We do as if the elements to be sorted were given one by one, the first element constituting, by itself, a sorted list of length 1. We then put away the second element to constitute a sorted list of length 2, then we put away the third element to have a sorted list of length 3 and so on....

The principle of sorting by insertion is to insert at the nth iteration the nth element at the right place.

3.1 In Place

Algorithm

```
def insert_sort2(L:list) -> None:
    """
    Sort a list using the insert sort algorithm
    (not in place)
    Parameters
    ----------
    L : List
        unsorted list.

    Returns
    -------
    Nothing.
    """
    n = len(L)
    for i in range(1, n):
        x = L[i]
        j = i
        while (j>0 and L[j-1] > x):
            L[j] = L[j-1]
            j -= 1
        L[j] = x
```

Complexity

Number of comparisons : at worst / at best
Number of copies of elements : at worst / at best

3.2 Not In place

Algorithm

```python
def insert(x:int, L:list) -> None:
    """
    Insert the element x in the right place in a sorted list

    Parameters
    ----------
    x : int
        element that we want to add.
    L : list
        sorted list.

    Returns
    -------
    Nothing.
    """
    n = len(L)
    i = 0
    while (i < n and x > L[i]):
        i += 1
    L.append( None)
    for j in range(n,i,-1):
        L[j] = L[j-1]
    L[i] = x

def insert_sort1(L:list) -> None:
    """
    Sort a list using the insert sort algorithm
    (in place)
    Parameters
    ----------
    L : List
        unsorted list.

    Returns
    -------
    list.
    """
    res = []
    for elt in L:
        insert(elt,res)
    return res
```

Complexity

Number of comparisons : at worst / at best
Number of copies of elements : at worst / at best

3.3 In place using binary search

Complexity

Number of comparisons : at worst / at best
Number of copies of elements : at worst / at best

Algorithm

```
def __binarySerch_rec(L:list, x:int, left:int, right:int) -> int:
    '''
    L, left, right : the list L[left, right[ to be considered
    x : an element
    return the index of the element x in L[left, right[
        if it is present, or where it should be
    '''
    if left >= right : #on a pas trouve l'element
        return left
    else :
        mid = left+(right-left)//2
        #(left + right) // 2
        # sauf que si left + right > maxint => probleme
        if L[mid] == x :
            return mid
        if x < L[mid] :
            return __binarySerch_rec(L,x,left,mid)
        else :
            return __binarySerch_rec(L,x,mid+1,right)

def binarySerch(L:list, x:int) -> int:
    return __binarySerch_rec(L,x,0,len(L))

# version insert en place qui utilise binarySerch
def __insert(x:int,L:list,n:int) ->  None:
    '''
    x : element
    L : list
    n : 1 < n < len(L)
    insert x at its position in the sorted list L[0, n[
    '''
    # search position
    i = __binarySerch_rec(L,x,0,n)
    # shifts
    for j in range (n,i,-1):
        L[j] = L[j-1]
    L[i] = x

def insertSort(L:list) ->  None:
    for i in range(1,len(L)):
        __insert(L[i],L,i)
```

4 Bubble Sort

4.1 Bubble Sort

Explanation

The principle of bubble sort or sinking sort is to compare two by two the consecutive elements e_1 and e_2 of an array and to perform a permutation if $e_1 > e_2$. We continue to sort until there is no more permutation.

Complexity

Number of comparisons : at worst $= \frac{n(n-1)}{2}$ / at best $= (n-1)$
Number of copies of elements : at worst $= n(n-1)$ / at best $= 0$

Iterative algorithm

```
def bubble_sort(L:list) ->  None:
    """
    Sort a list using the bubble sort algorithm

    Parameters
    ----------
    L : List
        unsorted list.

    Returns
    -------
    Nothing.
    """
    change = 42
    n = len(L)
    while change != -1:
        change = -1
        for i in range(n-1):
            if L[i] > L[i+1]:
                L[i],L[i+1] = L[i+1],L[i]
                change = i+1
        n = change
```

4.2 Shaker Sort

Explanation

The "Shaker" sort, also known as the "Cocktail" sort, is identical to the bubble sort except that it changes direction with each pass. This is a slight improvement because it not only allows larger items to migrate to the end of the series but also allows smaller items to migrate to the beginning.

Complexity

Number of comparisons : at worst / at best
Number of copies of elements : at worst / at best

Algorithm

```
def shaker_sort(L:list) -> None:
    """
    Sort a list using the Shaker sort algorithm

    Parameters
    ----------
    L : List
        unsorted list.

    Returns
    -------
    Nothing.
    """
    change = 42
    start = 0
    n = len(L)
    while change != -1:
        change = -1
        for j in range(start,n-1):
            if L[j] > L[j+1]:
                L[j],L[j+1] = L[j+1],L[j]
                change = j+1
        if change != -1:
            end = change
            change = -1
            for j in range(n-1,start,-1):
                if L[j] < L[j-1]:
                    L[j],L[j-1] = L[j-1],L[j]
                    change = j-1
            start = change+1
```

5 Divide and Conquer

5.1 Merge Sort

Explanation

It is again a sorting algorithm according to the divide and conquer paradigm. The principle of the merge sort is as follows:

1. The list to be sorted is divided into two halves
2. We sort each of them.
3. We merge the two halves obtained to reconstitute the sorted list.

Complexity

Number of comparisons : at worst / at best
Number of copies of elements : at worst / at best

Algorithm

```
def split(L):
    """
    Seperate L in two lists of almost the same length and return them

    Parameters
    ----------
    L : List
        list of length > 1.

    Returns
    -------
    (L1,L2) : tuple of list.
    """

    m = len(L)
    L1 = []
    for i in range(m//2):
        L1.append(L[i])
    L2 = []
    for i in range(m//2):
        L2.append(L[i])
    return (L1,L2)
```

Algorithm

```
def merge(l1,l2):
    """
    merge L1 and L2 in one into one sorted list and return it

    Parameters
    ----------
    L1,L2 : List
        two sorted lists (in increasing order)

    Returns
    -------
    L : List
        sorted list
    """
    L = []
    len1 = len(L1)
    len2 = len(L2)
    r1 = 0
    r2 = 0
    while r1 < len1 and r2 < len2:
        if L1[r1] < L2[r1]:
            L.append(L1[r1])
            r1 += 1
        else:
            L.append(L2[r2])
            r2 += 1
    for i in range(r1,len1):
        L.append(L1[i])
    for i in range(r2,len2):
        L.append(L2[i])
    return L
```

Algorithm

```
def merge_sort(L):
   if len(L)<=1:
        return L
   else:
        (L1,L2) = split(L)
        L1 = merge_sort(L1)
        L2 = merge_sort(L2)
        return merge(L1,L2)
```

5.2 Quick Sort

Complexity

Number of comparisons : at worst / at best
Number of copies of elements : at worst / at best

Algorithm

6 Complexity Recap

	Copies : Au pire	Copies : Au mieux	Comparaisons : Au pire	Comparaisons : Au mieux
Select Sort	$2(n-1)$	0	$\frac{(n-1)n}{2}$	n
Insert Sort	$(n+1)$	1	1	n
Bubble Sort	$n(n-1)$	0	$\frac{n(n-1)}{2}$	$(n-1)$
Cocktail Sort				
Merge Sort	n		n	$\frac{n}{2}$