Lecture 12: Sorting

l.sort()

$O(N \log N)$

l_new = sorted(l)

Types of sort:

1. Selection sort
2. Bubble sort
3. Insertion sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Counting sort
8. Radix sort
9. Bucket sort

(38) Given an array, remove every element from array one by one; cost to remove is the sum of all elements in it. Find minimum cost to remove all elements.

```
A.sort(reverse=True)
ans = 0
for i in range(len(A)):
    ans += A[i] * (i+1)

return ans
```

(37) Find the minimum difference of an array

min. value is $\min(|A[i]-A[j]|)$

where i, j are distinct.

Brute Force

```
ans = INF

for i in range(n):
    for j in range(i+1, n):
        ans = min(ans, abs(A[i]-A[j]))

return ans
```

$O(N^2)$

[1, -5, 3, 5, -10, 4]

⇒ 1

[-10 -5 1 3 4 5]

5 6 2 1 1

Optimized

```
n = len(A)
A.sort()
ans = INF
for i in range(n-1):
    ans = min(ans, A[i+1] - A[i])
return ans
```

$O(N \log N)$

(38) Given an array of N distinct elements, find count of Noble elements

[1 -5 3 5 -10 4]

2 1 3 5 0 4

⇒ 3

Brute Force

→ go to every no.
→ counts less than no. == no. itself

Optimized

```
l.sort()
cnt = 0
for i in range(len(l)):
    if A[i] == i:
        cnt += 1
return cnt
```

(39) Noble elements when elements are not distinct.

[-10 1 1 3 100]

0 1 1 3 4

```
l.sort()
check = [0]*n
for i in range(n):
    if l[i] = l[i-1]:
        check[i] = check[i-1]
    else
        check[i] = i
cnt = 0
for i in range(n):
    if l[i] == i:
        cnt += 1
return cnt
```

☆ Comparators and Key

(40) Sorting the array based on no. of factors

```
def count-factors(x):
    cnt = 0
    for i in range(1, x+1):
        if x % i == 0:
            cnt += 1
    return cnt
```

```
l = [12, 9, 1, 6, 7]
l.sort(key = count-factors)
```

→ [1, 7, 9, 6, 12]

1 2 3 4 6

```
def cmp(a,b):
    cnta = count-factors(a)
    cntb = count-factors(b)
    if cnta == cntb:
        return 0
    if cnta > cntb:
        return 1
    if cnta < cntb:
        return -1
```

-1 means it should be on the left

i/p [1, 7, 6, 9, 12]

```
from functools import cmp_to_key
l.sort(key = cmp_to_key(cmp))
```

→ [1, 7, 9, 6, 12]

1 will be compared to 7 first 1 < 7 returns -1

so, 1 will be before 7

1 ⟺ 6 1 before 6

& likewise for all elements to get final sorted list

comparators can be used for complex comparisons

Another example:

```
def descCmp(a,b):
    if a > b:
        return -1
    elif a < b:
        return 1
    return 0
```

-1 here means a should be on the left (over case a greater value on left)

```
l = [1, 2, 9, 12, 6]
l.sort(key = cmp_to_key(descCmp))
```

→ [12, 6, 9, 7, []]

(41) Given an array A of N integers. Arrange them in such a way that they form the largest number

[2, 3, 9, 0]

↳ 9320

```
from functools import cmp_to_key
def compare(a, b):
    if a+b > b+a:
        return -1
    else:
        return 1
A.sort(key = cmp_to_key(compare))
if A[0] == "0":
    return 0
return "".join(A)
```

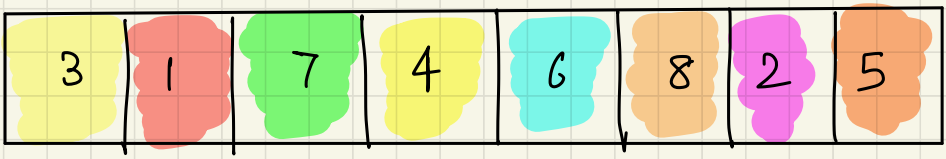
☆ Link to the blog :

https://towardsdatascience.com/5-sorting-algorithms-in-python-c7ece9df5dd6

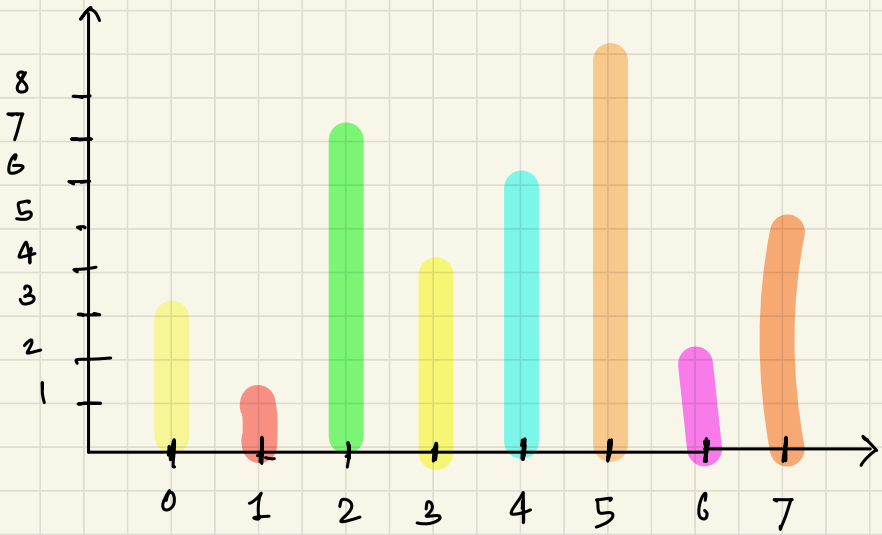
☆ Video for visualizing sorting algorithms

https://www.youtube.com/'watch?'v=kPRA0W1kECg'

Array Visualization



N = 8




```
def swap (arr, a, b):
    temp = arr[a]
    arr[a] = arr[b]
    arr[b] = temp
```

swaps elements at index "a" & "b"

1. Selection Sort

- in-place sorting algorithm, meaning the sorted items use the same storage as original elements

Ex. $[\underset{0}{1}, \underset{1}{4}, \underset{2}{-6}, \underset{3}{2}]$

i = 0 to 3 j = i+1 to 3

i = 0,

curr-min = 1
min-idx = 0

j : 1 ×

j : 2 curr-min > arr[2]
curr-min = -6
min-idx = 2

j : 3 ×
swap (1, -6) ⇒ [-6, 4, 1, 2]
(1 at index 0, -6 at index 2)

i = 1
& so on

```python
def selection_sort(self, unsorted, n):

    # iterate over array
    for i in range(0, n):

        # initialise with first value
        current_min = unsorted[i]

        # min_index initialiser
        min_index = i

        # iterate over remaining unsorted items
        for j in range(i, n):

            # check if jth value is less than current min
            if unsorted[j] < current_min:

                # update minimum value and index
                current_min = unsorted[j]
                min_index = j

        # swap ith and jth values
        swap(unsorted, i, min_index)
```

TC : $O(N^2)$

SC : $O(1)$

2. Bubble Sort

- bubble or sinking sort repeatedly passes over the list, comparing adjacent elements.
- Items are swapped depending on sorting condition

Ex. [64, 34, 25] (0, 1, 2)

i = 0 to 1 swapped = False

i = 0 j : 0 to 1

j = 0 64 > 34 → swap → [34, 64, 25] swapped = True

j = 1 64 > 25 → swap → [34, 25, 64] (n-1-1) 61

i = 1 j : 0 to 0

i = 1 j = 0 34 > 25 → swap → [25, 34, 64]

- larger values are bubbling to the end as program executes.

```python
def bubble_sort(self, unsorted, n):
    """ bubble sort algorithm """

    # iterate over unsorted array up until second last element
    for i in range(0, n - 1):

        # swapped conditions monitors for finalised list
        swapped = False

        # iterate over remaining unsorted items
        for j in range(0, n - 1 - i):

            # compare adjacent elements
            if unsorted[j].value > unsorted[j + 1].value:

                # swap elements
                swap(unsorted, j, j + 1)
                swapped = True

        # no swaps have occured so terminate
        if not swapped:
            break
```

TC:

Best case: when array is sorted

"n-1" comparisons $O(N)$

Worst case: sorting a descending array

$\frac{n * (n-1)}{2}$ comparisons $O(N^2)$

Average case: $O(N^2)$

SC: $O(1)$

3. Insertion sort

- builds the final array one item at a time.
- each object encountered on the outer loop is put between current closest minimum and maximum elements.

Ex. [9, 6, 7, 2, 5, 8]
0 1 2 3 4 5

i = 0 ×

i = 1 val = 6

hole = 1
arr[0] > val → arr[1] = 9

hole = 0
arr[0] = 6 [6 9 7 2 5 8]

i = 2 val = 7
hole = 2
arr[1] > 7 → arr[2] = 9
hole = 1
arr[0] > 7 ×
arr[1] = 7 [6 7 9 2 5 8]

& so on

```python
def insertion_sort(unsorted, n):
    """ insertion sort algorithm """

    # iterate over unsorted array
    for i in range(1, n):

        # get element value
        val = unsorted[i].value

        # insertion "hole" is at index i
        hole = i

        # loop backwards until a value greater than current value is found
        while hole > 0 and unsorted[hole - 1].value > val:

            # swap elements towards correct position
            unsorted[hole].value = unsorted[hole - 1].value

            # move backwards
            hole -= 1

        # insert value into correct position
        unsorted[hole].value = val
```

TC : $O(N^2)$

SC : $O(1)$