خوارزميات الترتيب

**Bubble Sort**

بنفس الطريقة التي ترتفع بها الفقاعات من أسفل الكأس،  bubble sort هو خوارزمية بسيطة لترتيب list ، عن طريق رفع أكبر قيم أو أصغر قيم إلى الأعلى مثل الفقاعات. الخوارزمية تمر (traverses) على list وتقارن القيم المتجاورة، وتقوم بتبديل أماكنهم اذا لم يكونوا بالترتيب الصحيح.

مع worst-case complexity of O(n^2) ، يعتبر bubble sort بطئ جدا عند مقارنته بخوارزميات اخرى مثل Quick Sort. ولكن على الجانب الاخر يعتبر من أبسط الخوارزميات لفهمه أو كتابته من الصفر.

من منظور تقني، bubble sort يستخدم في ترتيب مصفوفات صغيرة الحجم خاصة عند محاولة ترتيب مجموعة من العناصر على أجهزة كمبيوتر محدودة الذاكرة.

### مثال:

### أول دورة في المصفوفة:

* بداية من [4, 2, 6, 3, 9]، تقوم الخوارزمية بمقارنة أول عنصرين في المصفوفة، 4 و2 وتقوم بتبديلهم لأن 4>2 : [2, 4, 6, 3, 9]
* في الخطوة التالية تقوم بقارنة العنصرين التاليين، 4و6 وتجدهم بالفعل مرتبين وتبقى [2, 4, 6, 3, 9]
* العنصريين التاليين يتم تبديلهم لأن 6>3 : [2, 4, 3, 6, 9]
* العنصرين التاليين 6 و 9 بالفعل مرتبين لذا لا يتم تبديلهم

### ثاني دورة في المصفوفة:

* 2 < 4, وبالتالي لا حاجة لتبديل أماكنهم: [2, 4, 3, 6, 9]
* الخوارزمية تقوم بتبديل العنصرين التاليين لأن 3 < 4: [2, 3, 4, 6, 9]
* لا حاجة لتبديلات لأن 4 < 6: [2, 3, 4, 6, 9]
* , 6 < 9, وبالتالي مرة اخرى لا حاجة للتبديل: [2, 3, 4, 6, 9]  
  الـ List حاليا اصبحت مرتبة بالفعل، ولكن bubble sort لا يمكنه تمييز ذلك. وبالتالي يحتاج لإكمال دورة كاملة اخرى بدون اجراء أي تبديلات ليدرك أن الـ list مرتبة.

### ثالث دورة في المصفوفة:

* [2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]
* [2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]
* [2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]
* [2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]  
  من الواضح أن bubble sort بعيد جدا عن أن يكون أكفأ خوارزميات الترتيب. مع ذلك ، يمكنك بسهولة أن تنفذه بنفسك.

#### الخصائص

* Space complexity: O(1)
* Best case performance: O(n)
* Average case performance: O(n\*n)
* Worst case performance: O(n\*n)
* Stable: Yes 🡨 تحافظ الخوارزمية على الترتيب النسبي للعناصر التي تملك Keys متساوية. بعبارة أخرى، عند وجود عنصران متساويان فإنهم يحافظان على ترتيبهم بعد عملية الترتيب كما كانوا قبل عملية الترتيب (مثال:إذا كان (عنصر أ = عنصر ب) وعنصر أ يظهر في المجموعة الأصلية قبل الترتيب قبل عنصر ب فإنه يظهر في المجموعة النهائية بعد الترتيب قبل عنصر ب).

التنفيذ بلغة Python

def bubbleSort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1] :

arr[j], arr[j+1] = arr[j+1], arr[j]

print(arr)

**Merge Sort**

Merge Sort هي خوارزمية من نوع [Divide and Conquer](https://guide.freecodecamp.org/algorithms/divide-and-conquer-algorithms) .حيث يقوم بتقسيم المصفوفة إلى نصفين، ثم اعادة العملية مرة اخرى على كل من النصفين ثم إعادة دمجهم مرة اخرى بعد ترتيبهم. الجزء الرئيسي في الخوارزمية هو الجزء الخاص بدمج المصفوفتين المرتبتين في مصفوفة واحدة مرتبة. يمكن تلخيص عملية ترتيب مصفوفة تتكون من N من الأرقام الصحيحة (Integers) في ثلاث خطوات:

* تقسيم المصفوفة (array) الى نصفين.
* ترتيب النصفين الأيمن والأيسر بنفس الطريقة (recurring algorithm).
* دمج النصفين المرتبين.

هناك شئ يسمى بـ [Two Finger Algorithm](https://en.wikipedia.org/wiki/Cheney%27s_algorithm)  وهو الذي يعمل على دمج المصفوفتان المرتبتان معا. باستخدام هذا الـ subroutine و استدعاء merge sort function مع النصفين بشكل recursively نحصل على المصفوفة النهائية المرتبة التي نبحث عنها.

بما أن هذه الخوارزمية هي recursion based algorithm سنناقش الـ recurrence relation الخاص بها. الـ recurrence relation ببساطة هو طريقة بسيطة لتمثيل مشكلة معقدة اعتمادا على تقسيمها لعدة مشاكل أبسط. (subproblems)

T(n) = 2 \* T(n / 2) + O(n)

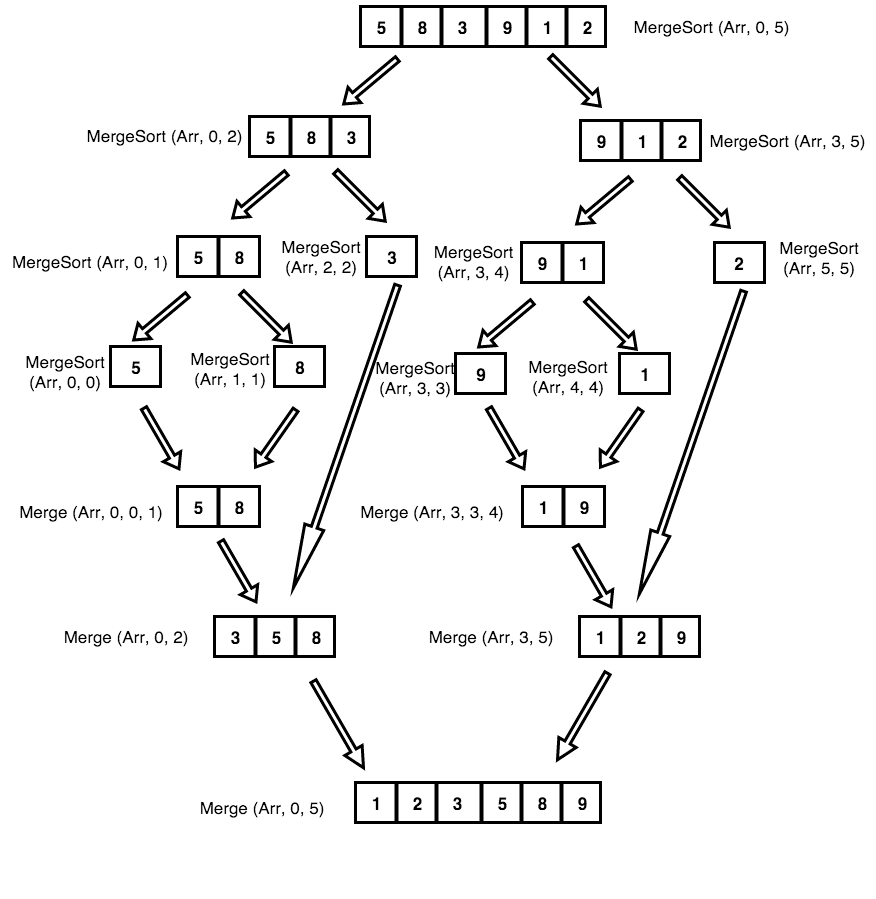
إذا أردنا التعبير عنها باللغة العربية، فهي ببساطة أننا نقوم بتقسيم المشكلة الى جزئين اصغر في كل خطوة و بعد ذلك نستطيع إعادة دمجهم مرة اخرى في نهاية كل خطوة مع linear amount of work .

### **التعقيد**

الميزة الأكبر من استخدام Merge Sort هي أن الـ [time complexity](https://www.youtube.com/watch?v=V42FBiohc6c&list=PL2_aWCzGMAwI9HK8YPVBjElbLbI3ufctn)  الخاص بها هو فقط n\*log(n) لترتيب المصفوفة بالكامل. وهو أفضل كثيرا من n^2 running time مع Bubble Sort أو Insertion Sort.

**مثال**

* في البداية نملك مصفوفة مكونة من 6 عناصر غير مرتبة Arr(5, 8, 3, 9, 1, 2)
* ثم نقوم بتقسيمها إلى نصفين Arr1 = (5, 8, 3) و Arr2 = (9, 1, 2).
* مرة اخرى نقسم كل array الى نصفين : Arr3 = (5, 8) و Arr4 = (3) و Arr5 = (9, 1) و Arr6 = (2)
* مرة اخرى نقوم بتقسيمها الى نصفين : Arr7 = (5), Arr8 = (8), Arr9 = (9), Arr10 = (1) و Arr6 = (2)
* الآن نقوم بمقارنة المصفوفات الجديدة معا لدمجهم مرة اخرى بطريقة مرتبة.



الخصائص:

* Space Complexity: O(n)
* Time Complexity: O(n\*log(n)). تقيد الزمان لـ Merge Sort يمكن أن تكون غير واضحة من أول نظرة . log(n) factor تم التعرف عليه من خلال ال recurrence relation المذكور بالأعلى.
* Sorting In Place: في التنفيذ الطبيعي، لا
* Stable: Yes
* متوازي: نعم (تمت مناقشة العديد من المتغيرات المتوازية في الطبعة الثالثة من مقدمة Cormen و Leiserson و Rivest و Stein للخوارزميات.)
* التنفيذ بلغة Python

def merge(left,right,compare):

result = []

i,j = 0,0

while (i < len(left) and j < len(right)):

if compare(left[i],right[j]):

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

while (i < len(left)):

result.append(left[i])

i += 1

while (j < len(right)):

result.append(right[j])

j += 1

return result

def merge\_sort(arr, compare = lambda x, y: x < y):

#Used lambda function to sort array in both(increasing and decresing) order.

#By default it sorts array in increasing order

if len(arr) < 2:

return arr[:]

else:

middle = len(arr) // 2

left = merge\_sort(arr[:middle], compare)

right = merge\_sort(arr[middle:], compare)

return merge(left, right, compare)

arr = [2,1,4,5,3]

print(merge\_sort(arr))