Sorting Unordered Lists: Shuttle Sort

Not all sorting algorithms are made equal. Bubble Sort, while being a good starting algorithm, is far from the most efficient solution out there. Crucially, its run time is proportional to n^2, which means if you go from a list of 1000 items to a list of 10,000, it'll take you 100 times longer to sort, rather than just 10 times longer. Shuttle Sort, also known as Insertion Sort, works on the principle of creating a small, sorted list, and gradually introducing new elements into the list, in the appropriate place.

Thought Experiment

Grab a deck of cards, give them a shuffle, and start picking cards from the deck one by one to add to your hand. If you're playing Go Fish, or Rummy, or some such game, you'll probably find it helpful to have your cards arranged in some sort of order. Usually, people will arrange their cards by rank, so that a hand might look like the following:

2, 3, 3, 5, 5, 5, 8, J, K, K

The question is, how do we generate this nicely sorted hand of cards? The answer is quite simple, really: one at a time:

Step 1: Draw your first card. It's a 5. Your hand is, trivially, in order.

Step 2: Draw a second card. This one's a 3, so, having compared to the 5, you move it to the left. You now have 2, 5. In order.

Step 3: Draw your third card. This one is a Jack, so having compared to the 5 you realize it belongs to the right of it, and you never bother comparing it to the 2, which is clearly smaller than the 5. You are taking advantage of the fact that your current list is sorted to make the process of adding items more efficient. You now have 2, 5, J.

Step 4: Continue. Get a 3? Compare to the Jack, and move it along. Compare to the 5, move it along. Compare to the 2, and stop – the 3 has found its place: 2, 3, 5, J.

Etc…

Give it a go

You are to sort the following list, following the same procedure as described above: 'pick up' the first item, and use it to start your list. Then pick up the next item, and compare, from right to left, with the items in your sorted sublist, stopping only when you reach its correct position. Insert it into your list, and repeat until all elements have been added to your list. Try to keep track of the number of comparisons you need to make, and the number of swaps (ie, movements past a number) you have to perform.

8 3 2 6 9 4 2 7

See the summary at the end to check your results.

Modelling it with Python

Pop

Since this algorithm involves creating a new list and gradually introducing new items to it, we will find it helpful to use the **pop** method, which both removes and returns (ie gives us) the relevant value from a list. See this example to understand the difference:

Using **remove** and **append** to transfer an element from one list to another:

```
1   beatles = ["John", "Paul", "George"]
2   drummers = ["Pete", "Ringo"]
3   new_guy = drummers[1]
4   drummers.remove(new_guy)
5   beatles.append(new_guy)
```

Using **pop** to do the same thing:

```
1   beatles = ["John", "Paul", "George"]
2   drummers = ["Pete", "Ringo"]
3   beatles.append(drummers.pop(1))
```

Both of these code snippets do the same thing: "Ringo" is removed from **drummers**, and attached to the end of **beatles**.

Range

We will also find it helpful to be able to walk backwards through a list, which is perfectly possible using additional optional arguments in a **range** object:

`for i in range(0, 3, 1):` `  print(i)`	The **range** object takes arguments **start**, **stop** and **step**.
`for i in range(0, 3):` `  print(i)`	If we only provide two numbers, they are assumed to be **start** and **stop** (and a default of 1 is used for **step**).
`for i in range(3):` `  print(i)`	If we only provide one number, it is assumed to be **stop** (and defaults of 0 and 1 are used for **start** and **step**).

*Recall: a range includes the **start** value, and goes up to <u>but not including</u> the **stop** value.*

How could we use a **range** object to cycle through every element in a list *from the last to the first?* Can you ensure that your code works regardless of the length of the list?

When you've had a go, compare to the code below:

```
1   my_list = [2, 3, 5, 7, 11, 13, 17, 19]
2
3   for i in range(len(my_list) - 1, -1, -1):
4     print(my_list[i])
```

Notice that we have to set the starting position to the length *minus one* because a list with 4 items is indexed from 0 to 3, etc.

We also need to give the stop as -1 to ensure the list includes index 0.
The step size is -1 as well, because we want to move down 1 each time.

Step by step…

As with the Bubble Sort algorithm, it will be easier to build up our algorithm step by step. We'll worry later about how to bring it all together, but for now, try to make a program that inserts a value into an already sorted list.

Inserting an element into a sorted list

My code is going to start by placing the new element at the end of the list, then, using a range to work backwards, successively compare pairs of elements, swapping if necessary, and breaking out of the loop (terminating the process) as soon as a swap is no longer required:

```
1   l = [2, 4, 7, 9] # sorted list
2   e = 5 # new element to insert
3   l.append(e)
4
5   for i in range(len(l) - 1, 0, -1):
6     if l[i] < l[i - 1]:
7       l[i], l[i - 1] = l[i - 1], l[i]
8     else:
9       break
```

We first append the new item to the end of our pre-sorted list.
Then, cycling through elements in the list backwards (**i** goes from 4 to 1 inclusive, so we compare items 4 & 3, then 3 & 2, then 2 & 1 and finally 1 & 0.
If a swap is required, it is done (using simultaneous assignment), and if not, the algorithm terminates – we're finished.

*Tip: don't be afraid to pepper your code with **print** statements, especially when you're just building it. It's the best way to identify exactly what's happening and find any bugs:*

```
5   print(l)
6   for i in range(len(l) - 1, 0, -1):
7     print(l)
8     print(f"Comparing {l[i]} with {l[i - 1]}...")
9     if l[i] < l[i - 1]:
10      print(f"{l[i]} is smaller.  Swapping...")
11      l[i], l[i - 1] = l[i - 1], l[i]
12    else:
13      print("No swap required.  Done.")
14      break
15  print(l)
```

Building the loop

Now we know how to insert a new element, shuttling it down the list until it reaches its appropriate position, we can start to generalise.

```
1   import random
2
3   original = [random.randint(1, 10) for i in range(10)]
4   print(original)
5
6   l = []
7   for e in original:
8     l.append(e)
9     for i in range(len(l) - 1, 0, -1):
10      if l[i] < l[i - 1]:
11        l[i], l[i - 1] = l[i - 1], l[i]
12      else:
13        break
14    print(l)
```

I'm using **random** to generate my original, unsorted, list.

I begin with an empty list, **l**, which has each element from the original list added one at a time. Each time a new element is added, it is shuttled into place. At the end of each shuttle, I print the list so I can check progress.

*Note that the first time this happens, the range object is **range(0, 0, -1)**, which contains nothing. This doesn't cause a problem – it just skips the loop entirely.*

Counting comps and swaps

In order to find out how well Shuttle Sort performs on different types of list, and how well it fares in direct competition with Bubble Sort, we'll need to build in a way to track comparisons and swaps. Look back at your Bubble Sort program and see if you can do something similar here.

```
4   def shuttle_sort(n):
5     a = [random.randint(1, 100) for i in range(n)]
6     total_swaps, total_comps = 0, 0
7     l = []
8     for e in a:
9       l.append(e)
10      for i in range(len(l) - 1, 0, -1):
11        total_comps += 1
12        if l[i] < l[i - 1]:
13          total_swaps += 1
14          l[i], l[i - 1] = l[i - 1], l[i]
15        else:
16          break
17    return total_comps, total_swaps
```

This function should look familiar – the general structure is taken from the Bubble Sort algorithm program we made, and it does the same thing: creates a random list of the specified length, sorts it (keeping track of swaps and comparisons along the way), and returns the results for analysis.

Reporting statistics

```
24  def print_stats(values, label):
25    print(label)
26    print(f" - Lowest: {min(values)}")
27    print(f"   - Mean: {round(sum(values) / len(values), 1)}")
28    print(f"     - Highest: {max(values)}")
29
30  trials = 1000
31  length = 100
32  print(f"Sorting {trials} random lists of size {length}...")
33  num_comps, num_swaps, costs = [], [], []
34  for i in range(trials):
35    c, s = shuttle_sort(length)
36    cost = s + 0.5 * c
37    num_comps.append(c)
38    num_swaps.append(s)
39    costs.append(cost)
40
41  print_stats(num_comps, "Comparisons:")
42  print_stats(num_swaps, "Swaps:")
43  print_stats(costs, "Costs:")
```

By reusing the code from Bubble Sort, we can do exactly the same thing. The only difference here is line 35, where I'm invoking the **shuttle_sort** function.

You should be able to do a direct comparison between the two algorithms by comparing the statistics produced by this code.

Even better if…

It's good to see the average and best / worst case scenarios for our simulations, but if we could plot the results on a histogram, the distribution would be even clearer. This is not too hard with **matplotlib.**pyplot, which is designed especially for this kind of task.

First, remember to import it at the top:

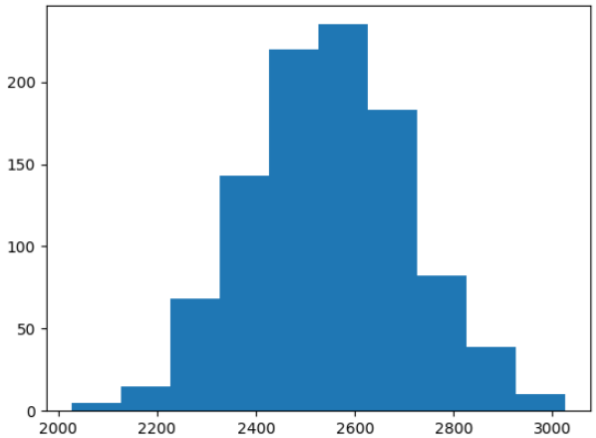
```
import matplotlib.pyplot as plt
```

Using **import … as …** lets us assign a neater label to this long-winded module name.

Then, at the bottom, write these two lines:

```
plt.hist(num_comps)
plt.savefig("comparisons.png")
```

This makes and saves the histogram. It may not show up on screen in repl.it, but it should appear as a .png file on the left.



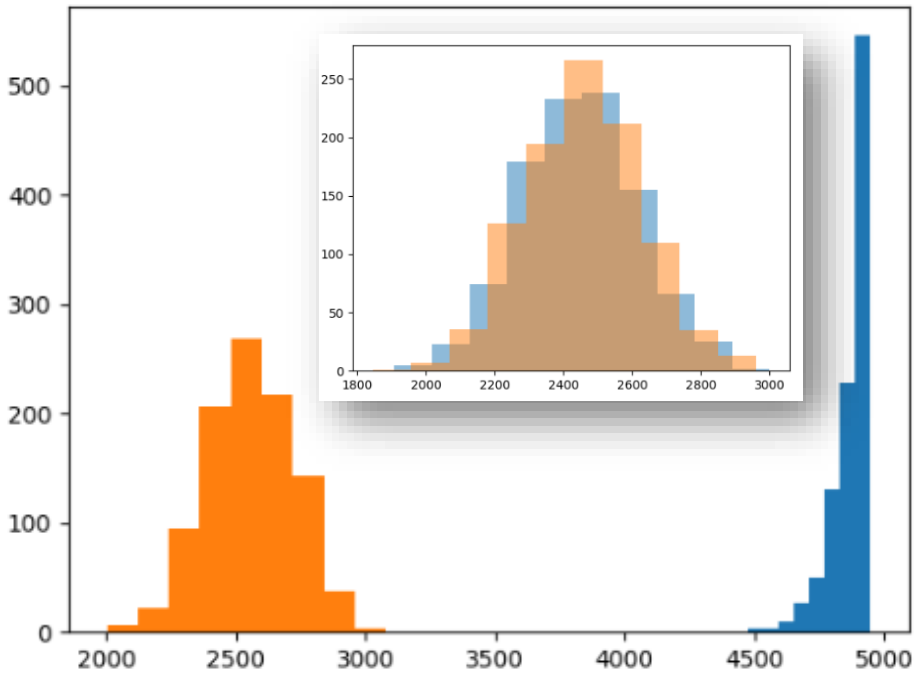
A mathematical comparison

We can use the same graphing code to explore the distribution for both sorting algorithms, and we can start to see which might come out on top. To see why, we'll have to do some good old-fashioned maths.

*Note: using **plt.hist** a second time in the same code will add your second histogram onto the same axes as the first. If they overlap, you might want to use **alpha** to make one or both of your histograms semi-transparent:* `plt.hist(num_swaps, alpha = 0.5)`

Number of Comparisons for random lists of length 100: Shuttle vs Bubble

Inset: Number of Swaps for random lists of length 100: Shuttle vs Bubble



The orange histogram on the left shows the number of comparisons for a shuttle sort.

The blue histogram on the right shows the number of comparisons for a bubble sort.

The inset histogram shows the number of swaps required for bubble sort (blue) and shuttle sort (orange).

According to our data, both sorting algorithms require around the same number of swaps (about 2500 for a 100-item list), but there is a marked difference in the number of comparisons required: around 2500 for Shuttle Sort, but almost twice that for Bubble Sort.

Both algorithms potentially take up to 4950 comparisons for a list of length 100, because in the case of a Bubble Sort, the first pass compares 99 pairs, the second 98, etc. And in the case of a Shuttle Sort, the first pass doesn't take any comparisons, but the second requires 1, the third 2, etc, up to the 100th requiring 99. In both cases, this gives:

$$\sum_{r=1}^{99} r = \frac{99}{2} \times (99 + 1) = 4950$$

However, this by no means tells the whole story. In the case of a Shuttle Sort, which theoretically could take 4950 comparisons, in general for a random list, the next item to be placed will, on average, find its position after only half as many comparisons (it's just as likely to be in the first half of the list you check as it is in the second, so on average, it'll only take $\frac{4950}{2} = 2475$. Bubble Sort has no such efficiencies, only reducing the number of comparisons if it happens to complete a full pass with no swaps. Even one large value near the start will force Bubble to make its full complement of passes and comparisons.

It would appear Shuttle Sort wins this round.

Some Extension Ideas

- Can you run some tests on very large lists, and see how each algorithm scales with the size of the list? What's the average number of comparisons for 1000 items, or 10,000 items, when using Shuttle compared to using Bubble?
- What is the effect of running each of these algorithms on a *nearly sorted list*? Can you write some code to generate a list which is in order apart from just one or two items? How do your algorithms compare in this case?
- How well do your algorithms perform in a list with many duplicate items? If you make a list of 1000 random one-digit numbers, does that significantly change the efficiency of the algorithm?
- Read up on the documentation for **matplotlib.pyplot** and see if you can make a neat display of the various charts, perhaps tiling them to show comparisons on the left, swaps on the right, with bubble sort on the top row, shuttle on the bottom? Can you layer line graphs to show how the average number of comparisons for each algorithm varies with the length of list?

Homework

Shuttle Sort is definitely an improvement on Bubble Sort, but – just like Bubble – it grows very quickly as the size of the list increases. If it takes 1 second to sort a list of all the names in a class, it'll take 9 seconds to sort a list of all the names in 3 classes, and if you want to sort all the names in a thousand classes it'll take a million seconds (a week and a half). While Shuttle will take, on average, half the time for each of these, the fact that it grows in proportion to the *square* of the list length is going to make it very hard to efficiently scale. It makes a lot of sense, therefore, that the next big thing in sorting algorithms makes use of an approach known as **divide and conquer**. Without any further details, see if you can come up with a method which might improve on the scalability of Shuttle and Bubble for very large lists. Even if you can't completely write up an algorithm that works (which is quite a tall order, to be fair), maybe you can come up with some ideas that have the potential to turn into a divide-and-conquer type algorithm.

It could be worse…

Finally, an introduction to sorting algorithms wouldn't be complete without an ironic nod-of-the-head to *bogosort*, a sorting algorithm which makes Bubble look tidy and efficient. See if you can work out how this one functions, and why it is so ludicrously bad.

```
1   import random
2
3   def in_order(l):
4     correct = [l[i] < l[i + 1] for i in range(len(l) - 1)]
5     return all(correct)
6
7   l = [random.randint(1, 10) for i in range(5)]
8   comps = 0
9   while True:
10    if in_order(l):
11      break
12    else:
13      comps += 1
14      random.shuffle(l)
15  print(f"Done (took {comps} comparisons).")
16  print(l)
```

Warning: if you try this one on any list with more than a very small number of items, you'll be waiting a while...

Additional info and how to write out an implementation of the Shuttle Sort algorithm:

Name:	A 'shuttle' moves a number all the way along a sublist until it gets to the correct position. Also known as 'insertion sort' since each subsequent item is inserted into position.
Summary:	Make an ordered list of the first two items, then insert subsequent numbers in their correct position within this list. Continue until all items have been shuttled into position.
Efficiency:	For nearly sorted lists: $O(n)$ (takes about n comparisons to add a new item to a sorted list) For reverse order lists (worst case): $O(n^2)$ (but generally more efficient than bubble sort) Good enough with small lists to be used as part of larger 'divide-and-conquer' algorithms.
Algorithm:	Compare the first 2 items and swap if needed. Compare the next item successively to items in the already sorted sublist (items 1 and 2), swapping to move down the list until it occupies the correct position (this sublist will always be in order, and gradually grows as more items are added). Repeat for the fourth item, etc, until all items are inserted.
Example:	Sort the list 8 3 2 6 9 4 2 7 using shuttle sort. `The list:` 8 3 2 6 9 4 2 7 `First pass:` 3 8 2 6 9 4 2 7 (comps: 1, swaps: 1) `Second pass:` 2 3 8 6 9 4 2 7 (comps: 2, swaps: 2) `Third pass:` 2 3 6 8 9 4 2 7 (comps: 2, swaps: 1) `Fourth pass:` 2 3 6 8 9 4 2 7 (comps: 1, swaps: 0) `Fifth pass:` 2 3 4 6 8 9 2 7 (comps: 4, swaps: 3) `Sixth pass:` 2 2 3 4 6 8 9 7 (comps: 6, swaps: 5) `Seventh pass:` 2 2 3 4 6 7 8 9 (comps: 3, swaps: 2) **Totals: comparisons: 19, swaps: 14** Note: When starting a new pass, underline in the previous list the sorted portion plus one additional item. Once a comparison shows a swap is not necessary, subsequent comparisons for that pass are not required. If an item shuttles to the very start, comparisons will equal swaps for that pass. Otherwise, there'll be one more than swaps. The algorithm concludes after every element has been inserted into the correct position.
Visual:	For more details and an animation of this algorithm for a number of different cases, see: www.sorting-algorithms.com