Sorting Unordered Lists: Shell Sort

One of the drawbacks of shuttle sort is that if a list has even a few items a long way from their correct position, it can take a long time to get them in their proper place. If we could find a way to get the list in approximately the right order before we apply the shuttle sort, that could be really useful…

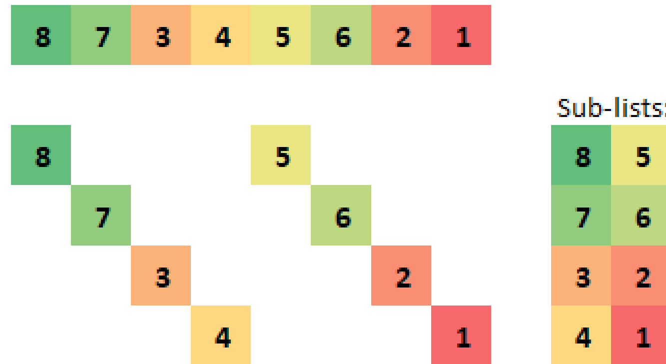
Investigate the problem

Consider the derangement shown below:

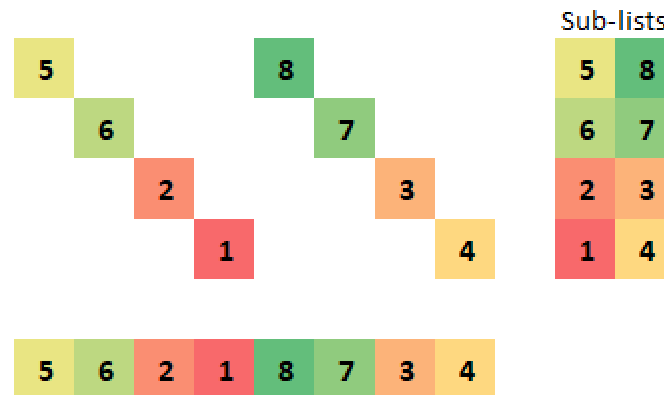
If we tried to sort this using shuttle sort, because the smallest values are at the wrong end, when we get to those values they'll need shuttling all the way through the list to the start.

The big idea, and the first step

The list is divided into a number of sub-lists, initially with just two items in each, but – importantly – far apart in the list. In the example above, for instance, items in positions 0 and 4 form a sub-list, and those in positions 1 and 5 form another, 2 and 6 another, and 3 and 7 the last. Each of these four sub-lists is sorted (using shuttle sort, but for now there are just two items in each, so it's pretty basic: compare and, if necessary, swap):



Once each of the four sub-lists of length 2 are sorted, the result looks like this:



Even with this modest adjustment (which required 4 comparisons and 4 swaps – worst case scenario for an 8 item list) we have turned our list from one which would require 25 comparisons and 22 swaps to one which only needs 19 comparisons and 14 swaps, a net saving of 2 comparisons and 4 swaps, making an overall cost saving of around 15% even on this fairly short list. If we take the worst case scenario for a shuttle sort (a reverse ordered list) and apply this process first, the efficiency saving is as high as 40%.

Try to code up this process, perhaps initially for an 8 item list, before reading on…

Running some tests

To see how effective this initial step of the algorithm is, we'll need to automate it, and try a few random lists. Notice how the sub-list positions are nice and regular. This should lend itself nicely to a **for** loop…

```
l = [1, 3, 5, 7, 8, 6, 4, 2]
for i in range(4):
    if l[i] > l[i + 4]:
        l[i], l[i + 4] = l[i + 4], l[i]
print(l)
```

This loop gives us the relevant sub-lists, and performs the comparison and, if necessary, swap, directly. Note that it only works on a list of length 8.

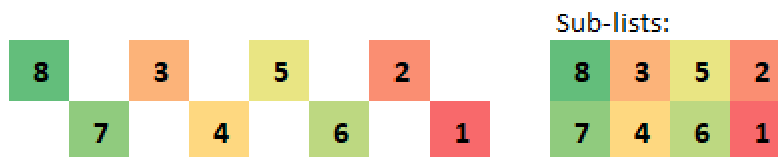
```
l = [1, 3, 5, 7, 8, 6, 4, 2, 12, 0]
n = len(l)
for i in range(n // 2):
    if l[i] > l[i + n // 2]:
        l[i], l[i + n // 2] = l[i + n // 2], l[i]
print(l)
```

This version generalises so it can deal with lists greater than length 8. In this case, it is working with a list of length 10, comparing values in positions 0 and 5, 1 and 6 etc.

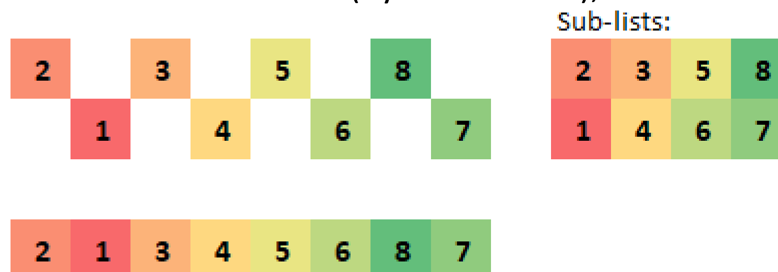
If we try this code out on a list with an odd number of terms, it still works, but some terms are left out. For instance, [5, 4, 3, 2, 1] becomes [3, 2, 5, 4, 1], leaving the 1 still at the far end. For this reason, the Shell sort algorithm in this form, and a number of other sorting algorithms, are at maximum efficiency when dealing with lists of length 2^n *for some* n*.*

Taking it to the next level

The process described is beneficial even on its own: it improves the cost of sorting 8 items by around 20% on average according to my simulations (sometimes as much as 60%, and sometimes (around 20% of the time) doing a bit worse than plain shuttle sort. These advantages start to peter out, however, the longer the list becomes, with a saving of not more than 1% on average for lists of length 100. To make our algorithm really powerful, we need to apply it, in modified form, repeatedly. The second step is to use sub-lists that are twice as long. Returning to our original example, it would look like this:



And, having sorted both of these sub-lists (by shuttle sort), we would end up with:



This is now looking somewhat more ordered. A final shuttle sort on this nearly-ordered list should have it in perfect order in 8 comparisons and 2 swaps. Added to the 6 comparisons and 5 swaps required for each of the two four-item shuttle sorts, we get 20 comparisons and 12 swaps, a cost improvement of almost 40%.

A note on efficiency

Shell sort is more complex in its approach than either Bubble or Shuttle, and it may well seem less efficient when being described, but that is mostly because we are developing our understanding of it primarily using smallish lists. Where it really comes into its own is for very long lists, where shuttle and bubble sort have a tendency to slow down significantly. For 8 items, applying steps 1 and 2 give an average improvement of 15% (although in some cases it can make the process significantly slower). As with step 1 alone, these two steps have less and less impact the longer the list grows. It is vital that we build in a way to repeatedly break the list into longer and longer sub-lists so that we can take full advantage of the method.

The second step, for a list of length n

We can make the second step with some clever list comprehension:

```
l = [2, 6, 4, 2, 7, 5, 3, 4]
n = len(l)
for i in range(n // 2):
    if l[i] > l[i + n // 2]:
        l[i], l[i + n // 2] = l[i + n // 2], l[i]
for i in range(n // 4):
    sublist = [l[i + n // 4 * k] for k in range(4)]
    sorted_sublist = shuttle_sort(sublist)
    for k in range(4):
        l[i + n // 4 * k] = sorted_sublist.pop(0)
print(l)
```

This starts out the same, using the generalised form for step 1. The next part uses **n // 4** (integer division by 4; that is, with no remainders) to find the number of sub-lists, then the list comprehension gives us the 0th, 2nd, 4th and 6th items on the first run through, and the 1st, 3rd, 5th and 7th positions on the next run through.

*Note: you'll need to create the **shuttle_sort** function separately, but you can adapt some of the code from the previous task. For now I'm just interested in the sorted list, so if you didn't care how it was done you could use Python's built-in method **sublist.sort()**, but soon we're going to want a way to track comparisons and swaps, which we'll collect info on from the **shuttle_sort** function.*

All the steps, for a list of length n

The real power of this algorithm comes from repeatedly building sub-lists, of ever-increasing size, shuttle sorting at each stage, until the 'sub-list' is the entire list. Shuttle sorting this (nearly ordered) list will then be significantly quicker than a straight shuttle. We have already seen how the first step can be effective for small lists. That efficiency saving is passed forward, because at each stage the sub-list being tackled by shuttle sort is 'nearly sorted' thanks to the previous step.

Edge cases

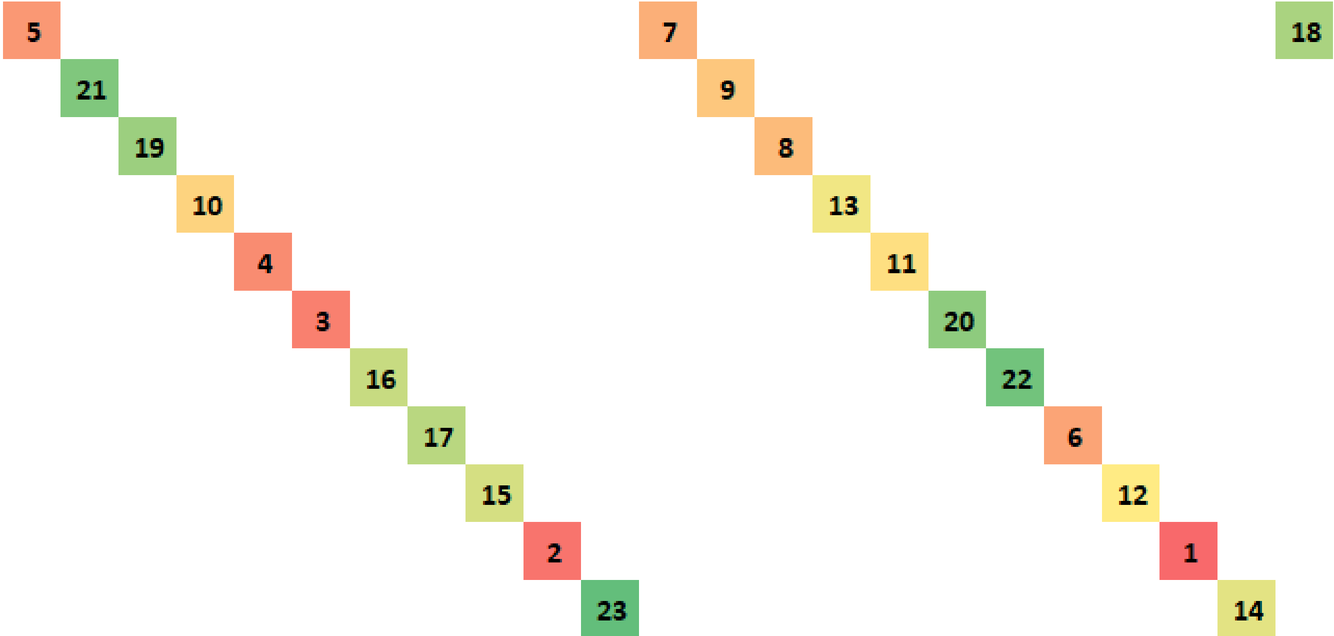
A proper Shell sort doesn't leave awkward extra terms out in the cold until the end. If we have 7 items to sort, the sub-lists will have a step size between elements of **7 // 2** initially, which means the sub-lists will contain items (0, 3, 6), (1, 4) and (2, 5) respectively. We will need to consider carefully how to code for this to ensure the full sub-list is created at each point, without going beyond the list (if you get an index error, you're referring to a position in a list which doesn't exist, eg asking for element 7 in a list of 7 (indices 0 to 6).

One more visual example before writing the final code

This example uses a list of length 23 to illustrate how halving the step (interval) between terms in the sub-lists works in practice, in particular for an awkward number of items:

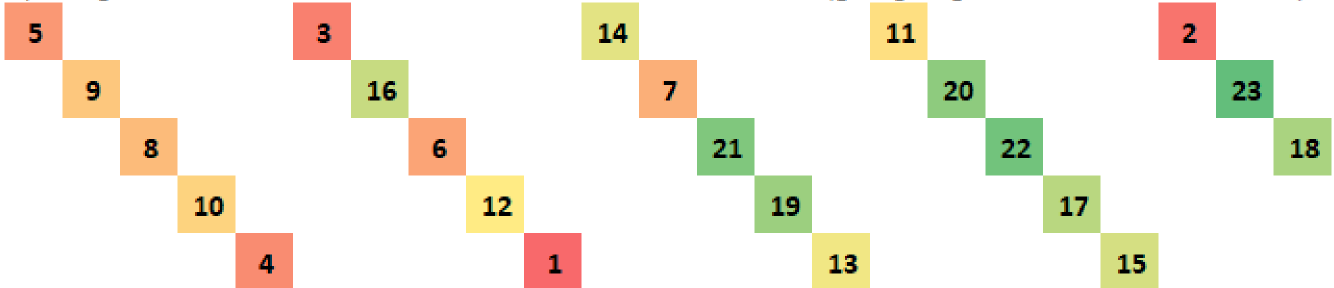
Original list:

Splitting into 23 // 2 = 11 sub-lists with an interval of 11 between terms (giving length 23 // 11 = 2 or 23 // 11 + 1 = 3):



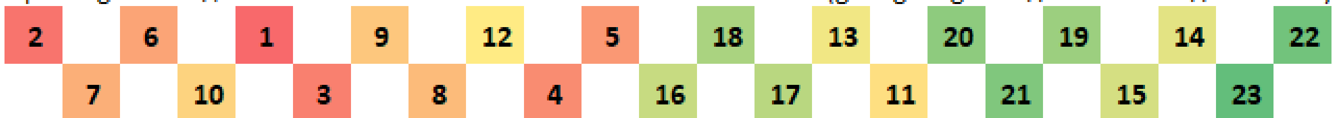
After sorting sub-lists (using shuttle sort for each)...

Splitting into 23 // 4 = 5 sub-lists with an interval of 5 between terms (giving length 23 // 5 = 4 or 23 // 5 + 1 = 5):



After sorting sub-lists (using shuttle sort for each)...

Splitting into 23 // 8 = 2 sub-lists with an interval of 2 between terms (giving length 23 // 2 = 11 or 23 // 2 + 1 = 12):



After sorting sub-lists (using shuttle sort for each)...

Splitting into 23 // 16 = 1 sub-list(s) with an interval of 1 between terms (giving length 23 // 1 = 23 or 23 // 1 + 1 = 24):

After sorting sub-lists (using shuttle sort for each)...

Notice how, at each stage, there may be sub-lists with one more item than others, due to the awkward nature of the number 23. Notice also how close to sorted we are by step 4.

Building the full algorithm with Python

python 3

Making sub-lists

Let's focus on the most pressing problem: for a list like the one we just looked at, with 23 elements, the length of each sub-list is not exact. But if we look at what *is* fixed, we might start to notice how similar this structure is to the **range** function. We know where each sub-list starts, and we know the value which it should not go beyond. We also know the step size to make. Let's test this out with some quick list comprehensions:

```
1   l = [0, 1, 2, 3, 4, 5, 6]
2   n = len(l)
3   step = n // 2
4   sublist = [l[i] for i in range(0, n, step)]
5   print(sublist)
6   sublist = [l[i] for i in range(1, n, step)]
7   print(sublist)
8   sublist = [l[i] for i in range(2, n, step)]
9   print(sublist)
10
11  for k in range(n // 2):
12      sublist = [l[i] for i in range(k, n, step)]
13      print(sublist)
```

I'm using a list of length 7, and I build a list comprehension to make use of the range function.

step should be 3, because **n** is 7. The first range ought to give 0, 3 and 6 (up in 3s from 0 to 7, not including 7).
The second range gives 1, 4 (since it doesn't include 7). Etc.

Remember that **range(start, stop, step)** generates values beginning with **start**, going up to (but not including) **stop**, adding **step** each time.

Getting the overall structure

It helps with any complex problem to break it down into stages. The same process that helps us lend structure to a long confusing maths problem is equally important when programming. Assuming we may use our code within other programs to sort lists and perform tests, we'll want to write a function (call it **shell_sort**) that will produce for us both a sorted list and the total number of comparisons and swaps required to sort it:

- Define **n**, and **step**, the length of the list and the initial step size for sub-lists.
- Create a loop to perform the splitting into sub-lists, sorting and tracking comparisons and swaps. This could continue, halving **step** each time, till **step** is 1.
    - Within the loop, use the range idea above to loop through sub-lists.
        - For each sub-list, invoke a **shuttle_sort** function to both sort the list and provide information about the number of comparisons and swaps.
        - Update the original list with the values of the sorted sub-list.
    - Update a tally of comparisons and swaps for each iteration through the loop.
- Return the now fully sorted list, along with the final tally of comparisons and swaps.

Give it a go

Using the structure above, try to create a program that performs a Shell sort. As a test case, give it the list shown on the previous page:

[5, 21, 19, 10, 4, 3, 16, 17, 15, 2, 23, 7, 9, 8, 13, 11, 20, 22, 6, 12, 1, 14, 18]

If you get it working properly, it should require exactly **107** comparisons and **47** swaps.
Protip: liberally scatter your code with print statements to check the values of different variables, to see exactly what sub-lists are being created, and make debugging easier.

Setting the scene

I want to use the structure of our previous sorting functions as far as possible, to improve readability and consistency. While the main priority is making code that works, if you aim to produce clear, well structured and readable code right from the start, not only will you have produced something more adaptable and easier to extend or debug later, but you'll find that the process of making your code work in the first place is that much easier.

```
1   import random
2   import matplotlib.pyplot as plt
3
4   def shuttle_sort(a):
5     total_comps, total_swaps = 0, 0
6     l = []
7     for e in a:
8       l.append(e)
9       for i in range(len(l) - 1, 0, -1):
10        total_comps += 1
11        if l[i] < l[i - 1]:
12          total_swaps += 1
13          l[i], l[i - 1] = l[i - 1], l[i]
14        else:
15          break
16    return l, total_comps, total_swaps
```

This code is taken from the previous program on the Shuttle Sort algorithm. We'll want the **random** module and **matplotlib.pyplot** for later, and we need the **shuttle_sort** function so we can call it from within our **shell_sort** function.

I am also going to use this function to model the format of **shell_sort**, tracking comparisons and swaps in the same way, and returning a tuple containing the sorted list, **l**, along with the number of comparisons and swaps required to sort it.

Writing the full shell sort program

Now for the code itself. By following carefully our bullet-point plan (and amending or adding to it as you run into snags or find the complexity too much), we should be able to put the whole thing together, and then test it with the example used earlier:

```
18  def shell_sort(a):
19    total_comps, total_swaps = 0, 0
20    l = a.copy()
21    n = len(l)
22    step = n // 2
23    while step >= 1:
24      for k in range(step):
25        sublist = [l[i] for i in range(k, n, step)]
26        shuttled, comps, swaps = shuttle_sort(sublist)
27        total_comps += comps
28        total_swaps += swaps
29        for i in range(k, n, step):
30          l[i] = shuttled.pop(0)
31      step //= 2
32    return l, total_comps, total_swaps
```

We track comparisons and swaps throughout, adding any that the shuttle sort performed on line 25 tells us were necessary.

Recall that **step** is not only the gap between terms in each sub-list, but also the number of sub-lists.

I use **pop(0)** to both return the first item in the sorted sub-list and remove it from that sub-list.

Note: with shuttle sort we built the sorted list up from an empty one, gradually adding items read from the original list provided. Since shell sort acts on the list given, a potential side-effect of running this program is that not only is a sorted version of the list produced as one of the function's return values, but the original list *has now been changed. This isn't always desirable, so I've used the* ***.copy()*** *method to produce a copy, leaving* ***l*** *in tact.*

Run some tests

Let's build another function that can quickly generate random lists and sort them using the function you created, first clarifying our requirements:

- I'd like Python to produce lists of random elements with a length of my choosing.
- I want to be able to generate as many of these as I like, and **shell_sort** them.
- I would like to receive as my output information a list containing the number of comparisons, and another list containing the number of swaps, required on all tests.

```
34  def random_test(num_lists, length):
35      comparisons, swaps = [], []
36      for i in range(num_lists):
37          l = [random.randint(1, 100) for i in range(length)]
38          _, c, s = shell_sort(l)
39          comparisons.append(c)
40          swaps.append(s)
41      return comparisons, swaps
42
43  c, s = random_test(10, 23)
44  print(f"Comparisons: {c} (average: {sum(c) / len(c)})")
45  print(f"Swaps: {s} (average: {sum(s) / len(s)})")
```

We make lists into which we can put the number of comparisons and swaps for each run.

List comprehensions let us create our random list in a single line.

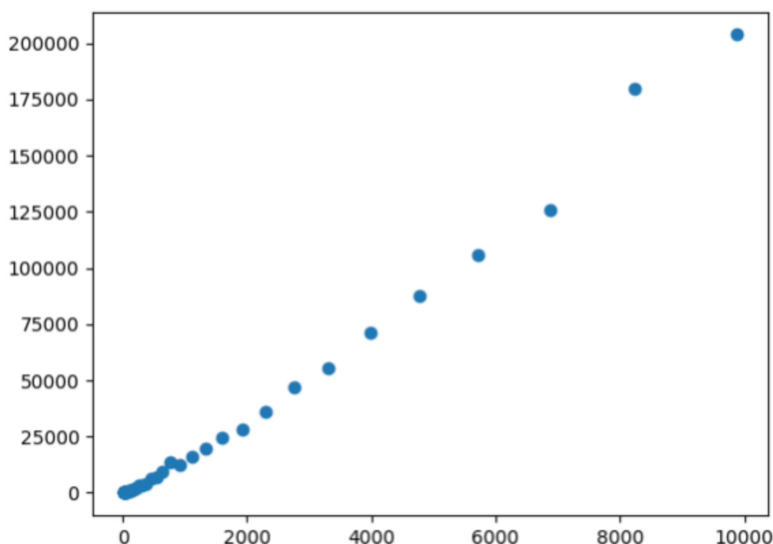
The underscore _ on line 38 lets us discard an unneeded element from a tuple. It tells Python: "Unpack the 3 items of the tuple produced, but throw away the first one".

How well does it scale?

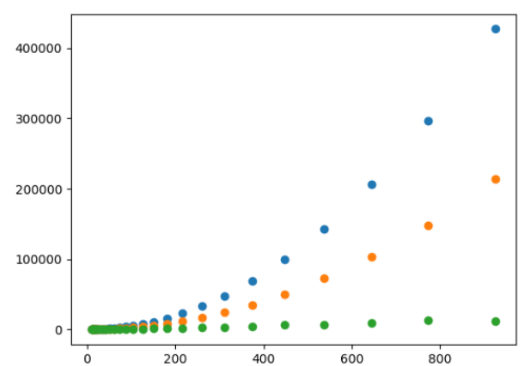
The million-dollar question. Both bubble and shuttle are $O(n^2)$, which is coder-speak for "cost is proportional to the square of the size of problem". How well does Shell Sort do?

```
43  lengths, comps = [], []
44  length = 10
45  while length <= 10000:
46      c, _ = random_test(10, length)
47      print(f"For a random list of length {length}:")
48      print(f"Average comps: {sum(c) / len(c)}")
49      lengths.append(length)
50      comps.append(sum(c) / len(c))
51      length = int(length * 1.2)
52
53  plt.scatter(lengths, comps)
54  plt.savefig("Comparisons vs Length Shell Sort.png")
```

There are plenty of ways to run this comparison, but, after a bit of trial and error, I decided I liked this one the best: it shows me information for small lists, then lists 20% longer, and so on, until it reaches very long lists.
Then it uses **matplotlib.pyplot** to show the trend on a graph.



This looks almost linear!
It's more striking when plotted along with the corresponding results from Bubble and Shuttle:



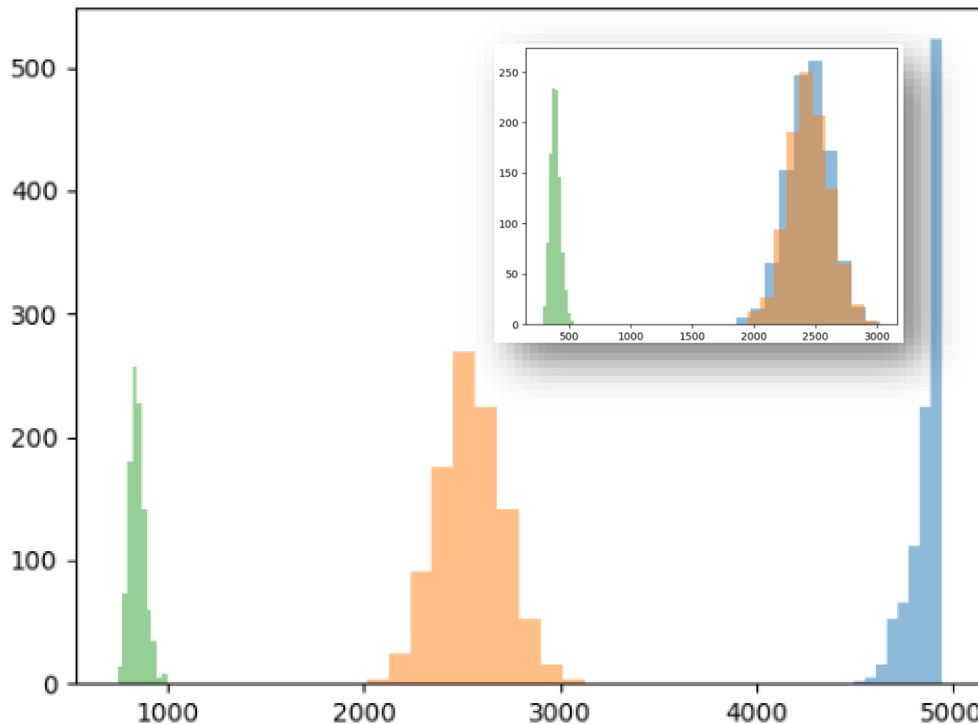
In conclusion

The Shell Sort has shown itself to be massively superior to both Bubble and Shuttle. It turns out that the worst case scenario (perfect reverse order) still takes the same number of swaps and comparisons as bubble or shuttle (although, unlike Bubble, which is often close to worst-case performance, Shell sort usually performs significantly better. In fact, this algorithm, believe it or not, is *still* not guaranteed to do any better than $O(n^2)$, although on average it is somewhere between linear and quadratic. It certainly appears fairly linear for our random lists of length up to 10,000 or so in our graph above, although without a more detailed analysis it is not clear if this is truly linear or just a very flat quadratic. With some small tweaks (see below) it can be improved still further to $O\left(n^{\frac{4}{3}}\right)$.

Even better if…

You could use the **scipy.stats** module to perform a statistical analysis on the data you create. Is it really a linear curve, or is it just a very gradual polynomial?

You could use the **hist** method from **matplotlib.pyplot** to overlay Shell Sort data on top of your Bubble and Shuttle Sort data.



The main plot shows the number of comparisons required to sort 1000 random lists of length 100 for each method: Shell, Shuttle, Bubble

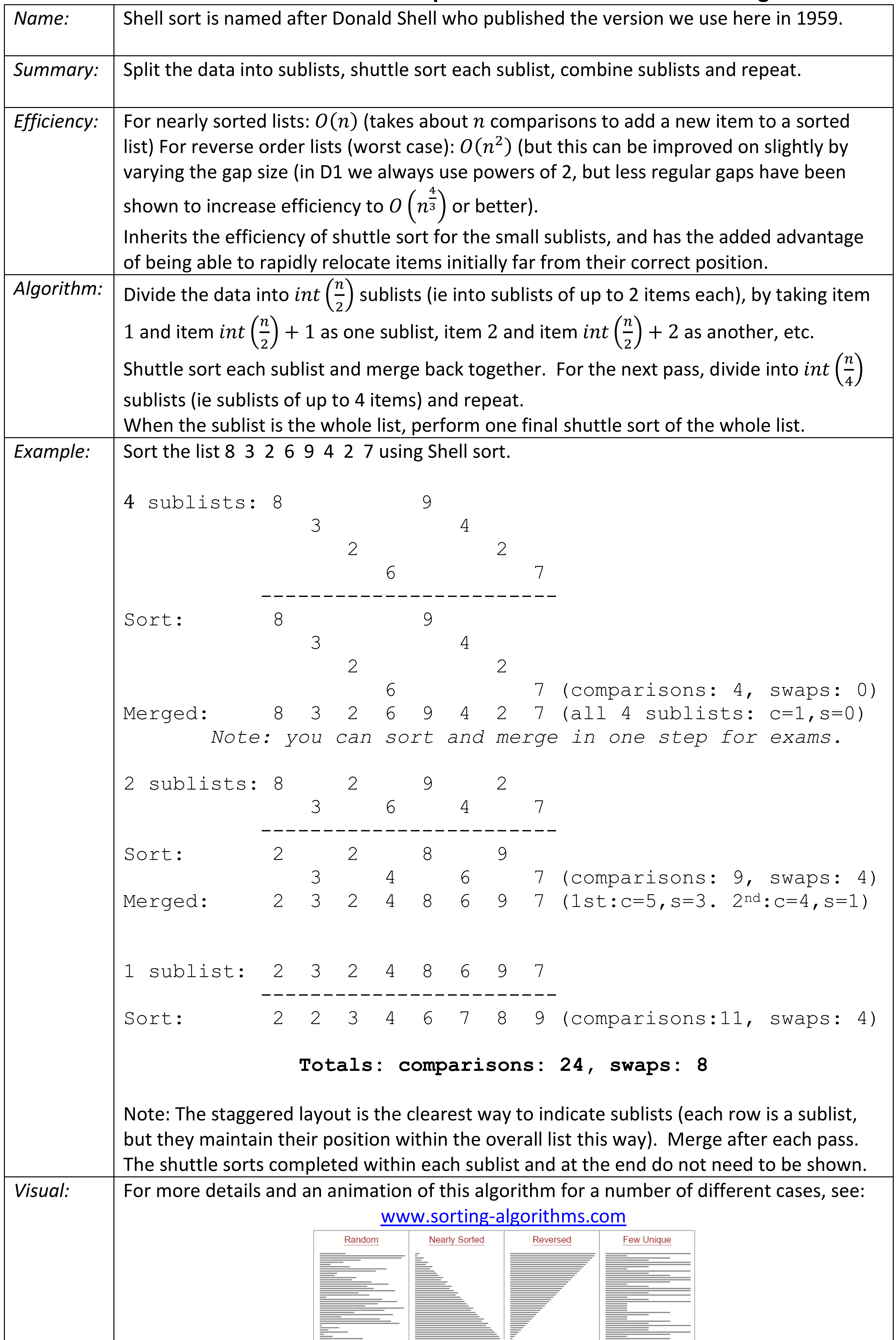
Inset, the smaller histogram shows a comparison of swaps required for each.

You could even try a different approach to creating your sub-lists. The basic principle of sorting lots of smaller lists, then fewer larger lists, and so on, can be riffed on somewhat, and it turns out, if you get creative with how you select elements for the sub-lists, you can improve the efficiency still further. See the Wikipedia article on the Shell sort and read the information under 'gap sequences' for more information on this.

What's next?

Is it possible to improve still further? We need at least $n-1$ comparisons just to check if a list of length n is sorted, so even the best possible algorithm can't get better than that. Shell sort is about 10 times quicker than bubble sort on 100 items, and 10 times quicker than that would mean an average of 50 comparisons, which is impossible. But even small gains can add up when lists are very long. Can you come up with your own original idea?

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



Name:	Shell sort is named after Donald Shell who published the version we use here in 1959.
Summary:	Split the data into sublists, shuttle sort each sublist, combine sublists and repeat.
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 this can be improved on slightly by varying the gap size (in D1 we always use powers of 2, but less regular gaps have been shown to increase efficiency to $O\left(n^{\frac{4}{3}}\right)$ or better). Inherits the efficiency of shuttle sort for the small sublists, and has the added advantage of being able to rapidly relocate items initially far from their correct position.
Algorithm:	Divide the data into $int\left(\frac{n}{2}\right)$ sublists (ie into sublists of up to 2 items each), by taking item 1 and item $int\left(\frac{n}{2}\right)+1$ as one sublist, item 2 and item $int\left(\frac{n}{2}\right)+2$ as another, etc. Shuttle sort each sublist and merge back together. For the next pass, divide into $int\left(\frac{n}{4}\right)$ sublists (ie sublists of up to 4 items) and repeat. When the sublist is the whole list, perform one final shuttle sort of the whole list.

Example: Sort the list 8 3 2 6 9 4 2 7 using Shell sort.

```
4 sublists: 8           9
               3           4
                  2           2
                     6           7
            ------------------------
Sort:       8           9
               3           4
                  2           2
                     6           7 (comparisons: 4, swaps: 0)
Merged:     8  3  2  6  9  4  2  7 (all 4 sublists: c=1,s=0)
       Note: you can sort and merge in one step for exams.

2 sublists: 8     2     9     2
               3     6     4     7
            ------------------------
Sort:       2     2     8     9
               3     4     6     7 (comparisons: 9, swaps: 4)
Merged:     2  3  2  4  8  6  9  7 (1st:c=5,s=3. 2nd:c=4,s=1)


1 sublist:  2  3  2  4  8  6  9  7
            ------------------------
Sort:       2  2  3  4  6  7  8  9 (comparisons:11, swaps: 4)

            Totals: comparisons: 24, swaps: 8
```

Note: The staggered layout is the clearest way to indicate sublists (each row is a sublist, but they maintain their position within the overall list this way). Merge after each pass. The shuttle sorts completed within each sublist and at the end do not need to be shown.

Visual:	For more details and an animation of this algorithm for a number of different cases, see: www.sorting-algorithms.com Random Nearly Sorted Reversed Few Unique