BubbleSort

We are now looking at a few sorting algorithms, starting with the classical “Bubblesort”. We do so in implementing the algorithm in various ways to compare the efficiency of the implementations

```
Needs["SortingUtilities`"];
```

Classical implementation with Do-Loops

At first we implement bubble sort with a strictly procedural approach: two nested loops, in the inner loop we count the number of change-operations to set this in relation to the run time of the algorithm.

First the (straightforward) implementation

```
bubbleSort[list_List] :=
 Module[{l = list, i, j, len = Length[list], count = 0},
  Do[
   Do[
     If[l[[j - 1]] > l[[j]], {l[[j - 1]], l[[j]]} = {l[[j]], l[[j - 1]]};
       count = count + 1, True], (*count changes*)
     {j, len, i, -1}];
   , {i, 1, len - 1}];
  {l, count}
  ]
```

Lets look whether this works

```
list = RandomInteger[100, 20]
```

```
{56, 4, 32, 37, 42, 15, 72, 14, 100, 38, 3, 35, 7, 69, 32, 91, 10, 32, 97, 55}
```

```
bubbleSort[list]
```

```
{{3, 4, 7, 10, 14, 15, 32, 32, 32, 35, 37, 38, 42, 55, 56, 69, 72, 91, 97, 100}, 84}
```

If we calculate the timing of sorting a list with, e.g., 100 elements we get the following result (here we use the First function to get only the run time, because the result of Timing is a list consisting of the run time as a first element and the sorted list as a second one).

```
list = RandomInteger[200, 100];
```

```
RandomInteger[200, 100]; First @ Timing @ bubbleSort[list]
```

```
0.025324
```

To analyze the run time of the algorithm we create a bunch of lists of random integers with increasing length to be sorted with our algorithm. We make lists of length 50, 100, 150, ... 1000

```
testLists = Table[RandomInteger[2 i 100, i 50], {i, 1, 20}]; Length /@ testLists
```

```
{50, 100, 150, 200, 250, 300, 350, 400, 450,
 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000}
```

So we collect information about the number of exchange operations

```
exchangesBubbleSort =
  Table[{i 50, Last[ bubbleSort[testLists[[i]]]]}, {i, 1, 20}];
```

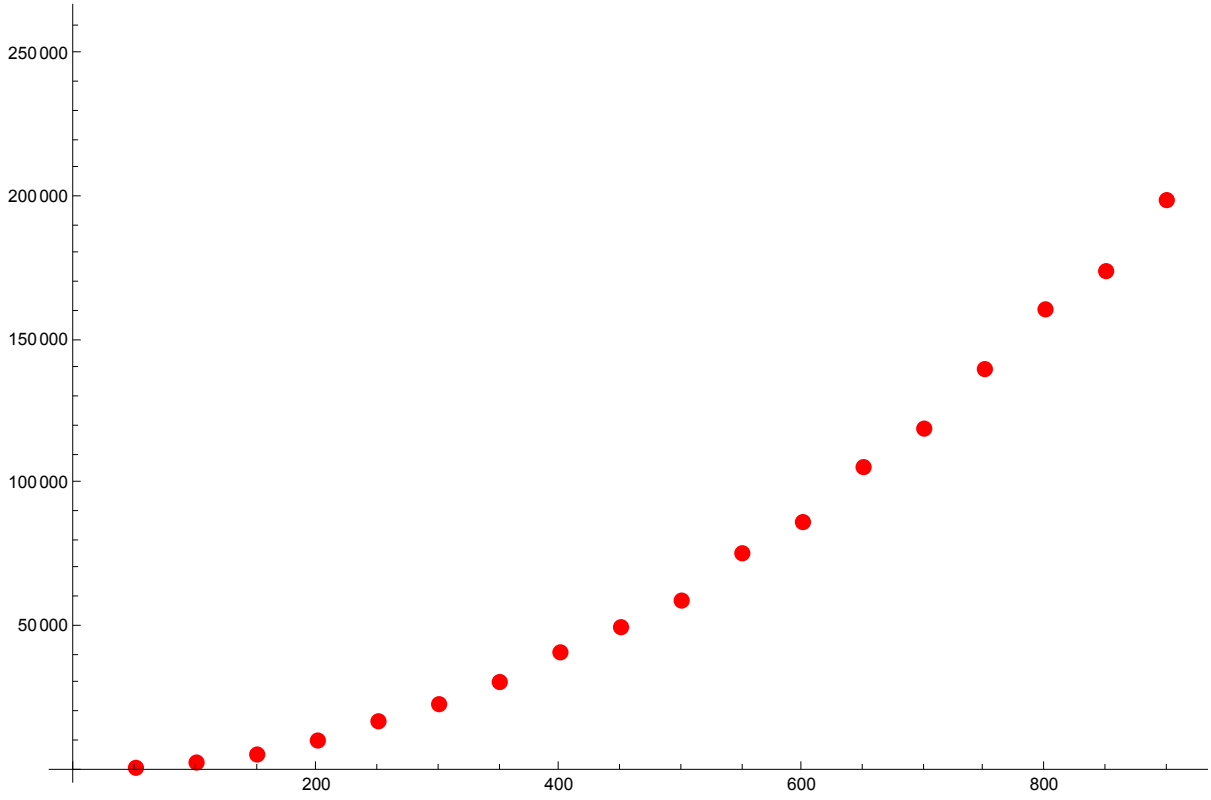
```
exchangesBubbleSort // TableForm
```

```
50      584
100     2428
150     5230
200     10 110
250     16 830
300     22 810
350     30 551
400     40 922
450     49 713
500     59 009
550     75 542
600     86 440
650     105 644
700     119 091
750     139 886
800     160 761
850     174 119
900     198 937
950     231 760
1000    253 735
```

What can immediately be seen is that the number of changes of neighbored elements seems to grow quadratically.

So we take a look at the data

```
lp = ListPlot[exchangesBubbleSort, PlotStyle -> Red]
```



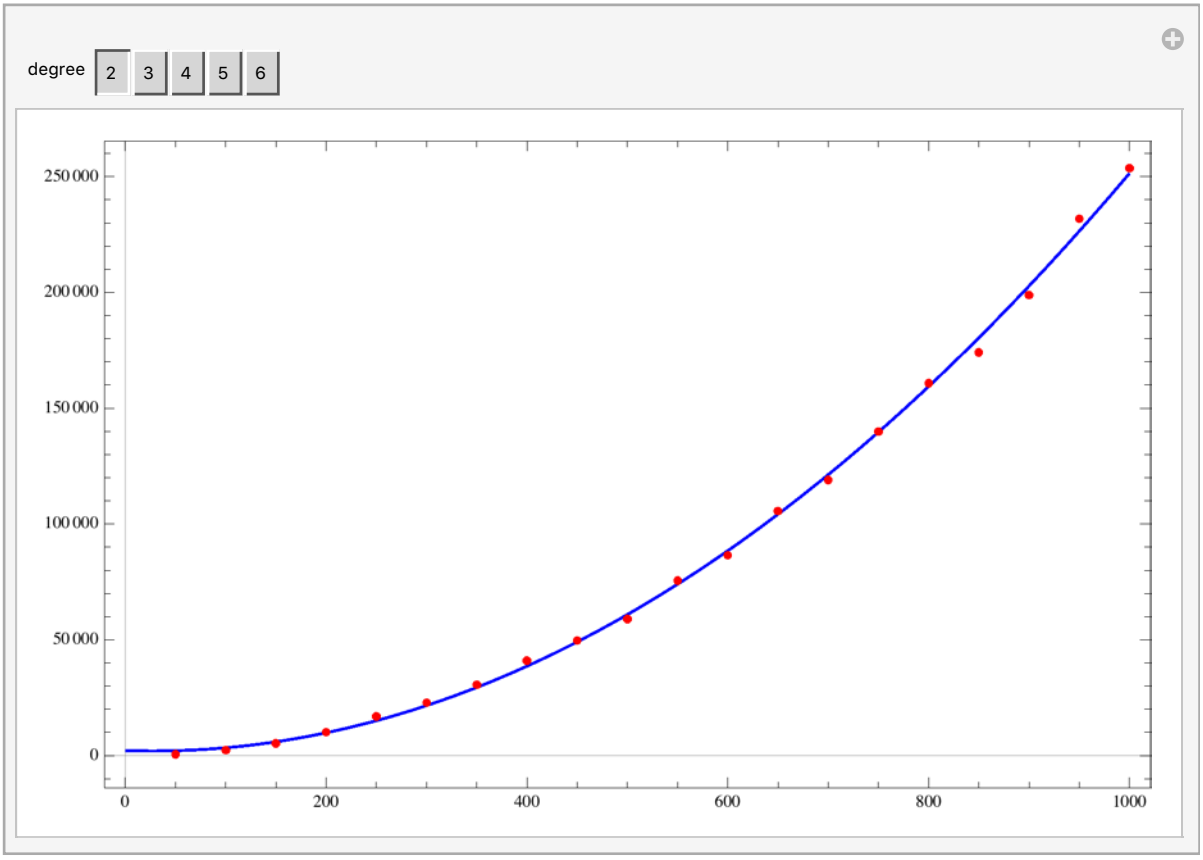
Now we are going to make a closer investigation of the dependence of the number of changes and the length of the corresponding list. Therefore we fit the data to polynomials of degree $n = 2$ up to $n = 6$

```
polynomials = Table[Fit[exchangesBubbleSort, Table[x^i, {i, 0, j}], x], {j, 2, 6}];
polynomials // TableForm
```

$$2183.31 - 14.4565\, x + 0.263725\, x^2$$
$$-2211.13 + 30.3894\, x + 0.15951\, x^2 + 0.0000661688\, x^3$$
$$149.361 - 6.54188\, x + 0.308975\, x^2 - 0.000151531\, x^3 + 1.03667 \times 10^{-7}\, x^4$$
$$-1586.03 + 30.4412\, x + 0.0857061\, x^2 + 0.00039355\, x^3 - 4.7251 \times 10^{-7}\, x^4 + 2.19496 \times 10^{-10}\, x^5$$
$$1243.14 - 45.5373\, x + 0.708622\, x^2 - 0.00182351\, x^3 + 3.36509 \times 10^{-6}\, x^4 - 2.95977 \times 10^{-9}\, x^5 + 1.0$$

It is obvious that the quadratic fit is the best of these polynomials, because the coefficients of the higher order Terms in the polynomials of degree greater than two are decreasing dramatically. But let us have a look at the graphical representation of the fitted polynomials

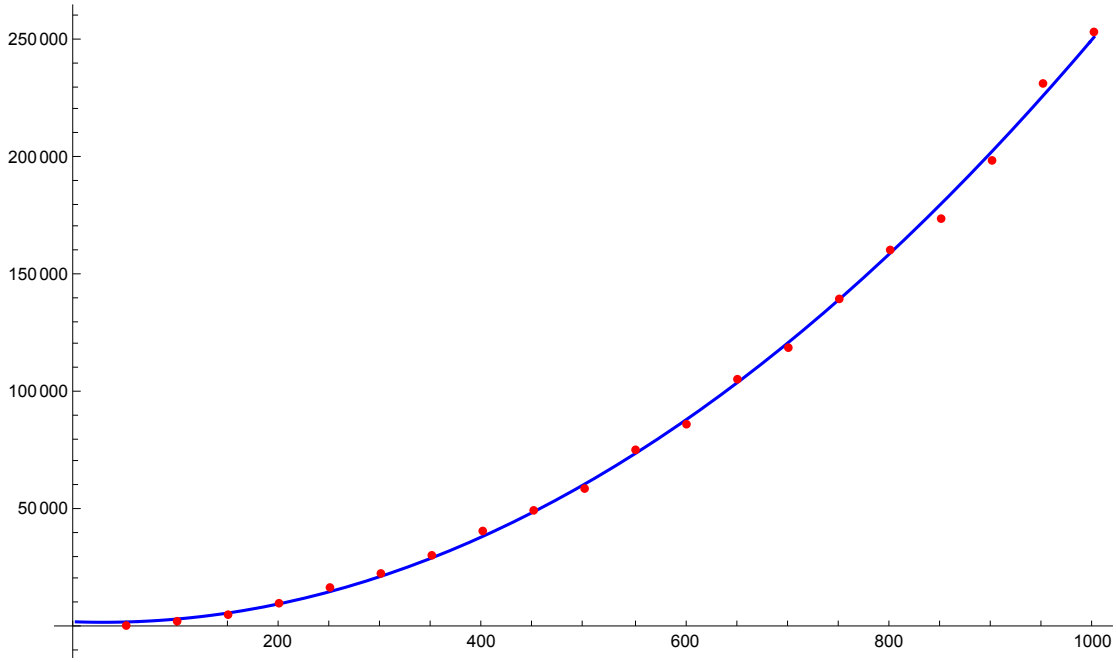
```
Manipulate[Plot[polynomials[[degree - 1]], {x, 0, 1000}, PlotStyle → Blue,
  Epilog → {Red, PointSize[Medium], Point /@ exchangesBubbleSort},
  ImageSize → Large], {degree, {2, 3, 4, 5, 6}}
]
```



So we see, that the approximation with a quadratic polynomial is quite good and, as suspected we have an $O(n^2)$ relation between the number of exchanges and the run time of the algorithm.

```
quadraticPolynomial = Fit[exchangesBubbleSort, {1, x, x^2}, x];
```

```
p = Plot[quadraticPolynomial, {x, 1, 1000}, PlotStyle → Blue, ImageSize → Large,
  Epilog → {Red, PointSize[Medium], Point /@ exchangesBubbleSort}]
```



Now for the analyzes of the run time, instead of the number of exchanges between neighbors. Therefore we calculate the run time for the sorting of our test lists. We therefore form a list of data pairs, consisting of the length of the list and the run time:

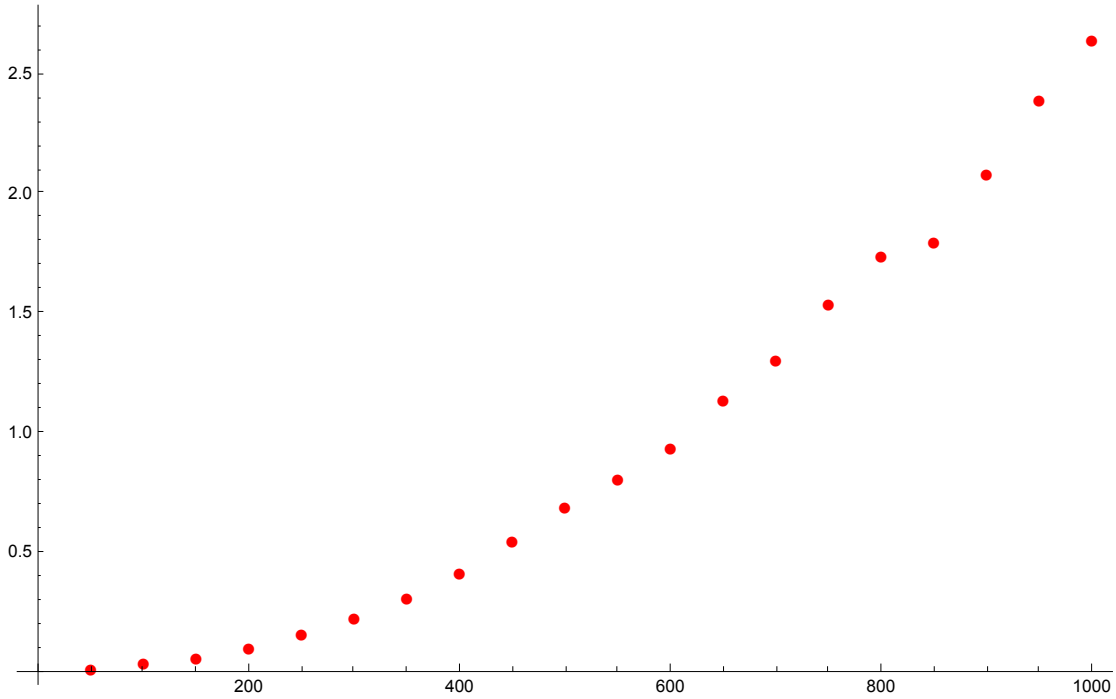
```
timingBubbleSort =
  Table[{i 50, First[Timing[ bubbleSort[testLists[[i]]]]]}, {i, 1, 20}];
```

```
Grid[Prepend[timingBubbleSort, {"Length", "Runtime"}],
 Dividers → {All, {1 → True, 2 -> True, -1 → True}}]
```

Length	Runtime
50	0.007616
100	0.031711
150	0.054623
200	0.094992
250	0.154386
300	0.220593
350	0.303035
400	0.409759
450	0.54388
500	0.685457
550	0.804012
600	0.930394
650	1.13375
700	1.30054
750	1.53567
800	1.73368
850	1.7944
900	2.07731
950	2.3875
1000	2.64034

From the data we have the strong evidence that there is also a quadratic relation between the run time and the length of the list.

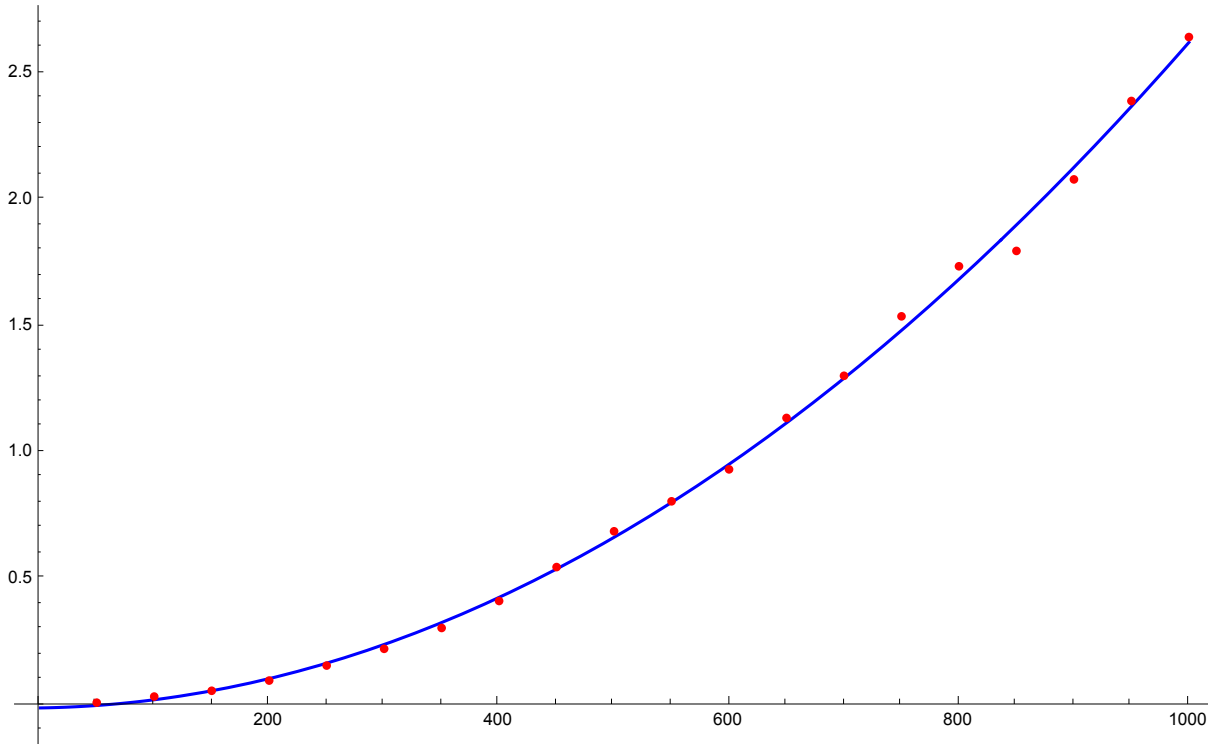
```
ListPlot[timingBubbleSort, PlotStyle -> Red,
 ImageSize → Large, PlotMarkers → {Automatic, Small}]
```



Now we again fit the data to a polynomial and choose a quadratic polynomial due to the obvious quadratic interdependence of the data

```
polyTime = Fit[timingBubbleSort, {1, x, x^2}, x];
```

```
Plot[polyTime, {x, 1, 1000}, PlotStyle → Blue,
 Epilog → {Red, PointSize[Medium], Point /@ timingBubbleSort}]
```



Implementation of Bubble-Sort with For-loops

Besides the Do-loops *Mathematica* also provides the (also classical) For-loops. So we implement the bubble sort algorithm now with For-loops and take a look whether the performance is affected by this implementational difference.

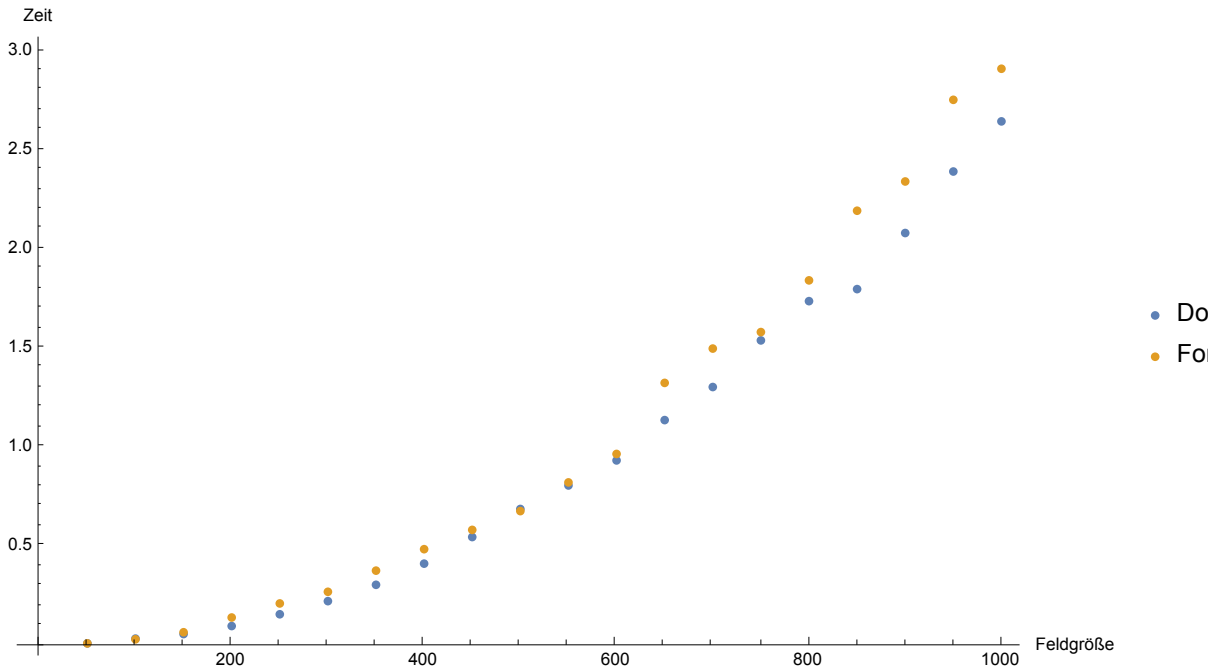
```
bubbleSortFor[list_List] :=
 Module[{l = list, i, j, len = Length[list], count = 0},
  For[i = 1, i <= len - 1, i++,
   For[j = len, j >= i, j--,
     If[l[[j - 1]] > l[[j]],
       {l[[j - 1]], l[[j]]} = {l[[j]], l[[j - 1]]}; count = count + 1, True]];
   ];
  {l, count}
  ]
```

Now lets calculate the data for the implementation with For-loops

```
timingBubbleSortFor =
  Table[{i 50, First[Timing[ bubbleSortFor[testLists[[i]]]]]}, {i, 1, 20}];
```

... and look at the results

```
ListPlot[{timingBubbleSort, timingBubbleSortFor},
 PlotStyle -> PointSize[Medium], PlotStyle → {Red, Blue},
 AxesLabel -> {"Feldgröße", "Zeit"},
 PlotLegends → {"Do", "For"}, ImageSize → Large]
```



Surprise, surprise! The implementation with Do-loops is always faster than an implementation with For-loops.

Rule-based Implementation

Another way to implement the bubble sort algorithm is a rule based implementation. The idea is to formulate a rule which says: “exchange two neighboring element if they are not in the right order”. This can be simply done with *Mathematica* in the following way (the notation **x___** here means zero or more occurrences of an element).

```
bubbleRule[xs___, x_, y_, ys___] := bubbleRule[xs, y, x, ys] /; x > y
```

This is a highly recursive approach (remember the number of exchange operations necessary to sort the list with bubble sort). Here is the result:

```
bubbleRule @@ list // Timing
```

```
{0.339066, bubbleRule[1, 4, 9, 10, 11, 11, 11, 11, 16, 19, 22, 23, 24, 25, 26, 27, 29,
  32, 32, 34, 36, 38, 40, 42, 50, 52, 55, 58, 59, 59, 61, 63, 67, 67, 68, 73, 75,
  77, 78, 82, 83, 90, 90, 92, 93, 93, 95, 96, 97, 99, 103, 104, 106, 106, 107,
  107, 109, 110, 111, 114, 114, 116, 118, 119, 120, 124, 127, 129, 129, 131,
  132, 134, 134, 136, 143, 143, 144, 145, 146, 147, 152, 152, 154, 157, 158,
  160, 161, 161, 162, 171, 172, 178, 181, 186, 186, 188, 191, 193, 196, 200]}
```

So we have about one third of a second (!) to sort 100 elements. Here is how this compares to the classical implementation we did before:

```
First[#] & /@ {Timing[bubbleSort[list]], Timing[bubbleRule @@ list]}
```

```
{0.024546, 0.318329}
```

Now let us test this with a larger list of, say, 200 elements

```
First @ Timing @ (bubbleRule @@ RandomInteger[1000, 200])
```

```
$IterationLimit::itlim: Iteration limit of 4096 exceeded. >>
```

```
1.19263
```

So we get an error message which says that the maximum limit of recursive function calls is exceeded (this limit is by default 4096). If we set this limit to a larger number, say ∞, we can sort the larger list

```
$IterationLimit = ∞; First @ Timing @ (bubbleRule @@ RandomInteger[1000, 200])
```

```
3.80312
```

So this is more than three seconds to sort the list! So, after resetting $IterationLimit to it´s original value, we look at the run time of this recursive approach:

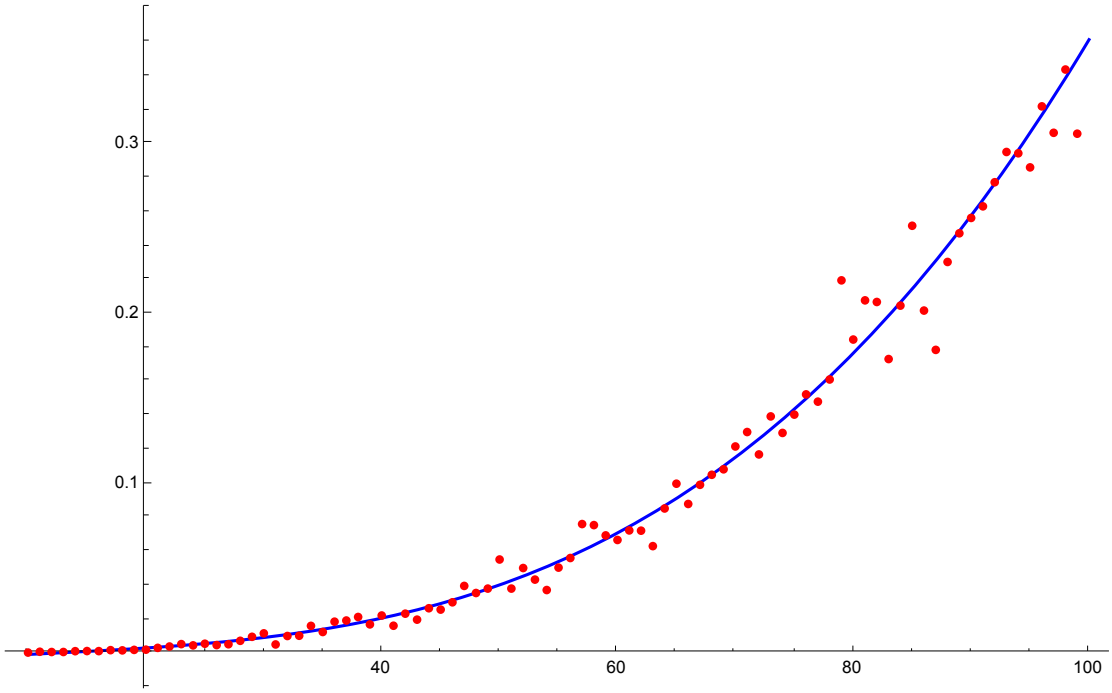
```
$IterationLimit = 4096;
```

```
timingBubbleSortRule = Table[
   {n, First @ (Timing[bubbleRule @@ RandomInteger[1000, n]])}, {n, 10, 100}];
```

```
guess = Fit[timingBubbleSortRule, {1, x, x^2, x^3}, x]
```

$$-0.00520761 + 0.000557479\,x - 0.0000171389\,x^2 + 4.82178 \times 10^{-7}\,x^3$$

```
Plot[guess, {x, 10, 100}, PlotStyle → Blue, ImageSize → Large,
 Epilog → {Red, PointSize[Medium], Point /@ timingBubbleSortRule}]
```



So we see that a complexity of $O(n^3)$ is quite a good approximation for this recursion based approach. But there is an other way to sort lists with a rule based approach. We can apply replacement rules to lists and do this also repeated. This avoids the drawback of massive recursion.

As an example we look at the sorting of a simple list with an replacement rule

```
RandomInteger[1000, 50] //. {a___, x_, y_, z___} /; x < y → {a, y, x, z}
```

```
{998, 980, 973, 968, 965, 953, 922, 884, 820, 791, 778, 760, 754, 745, 745, 742, 740,
 737, 730, 702, 694, 688, 688, 657, 644, 541, 540, 531, 521, 504, 487, 452, 440,
 434, 321, 307, 295, 229, 223, 221, 146, 107, 89, 87, 80, 73, 43, 34, 33, 29}
```

So we can define a rule for the implementation of the bubble sort

```
bubbleReplacementRule = {a___, x_, y_, z___} /; x < y → {a, y, x, z}
```

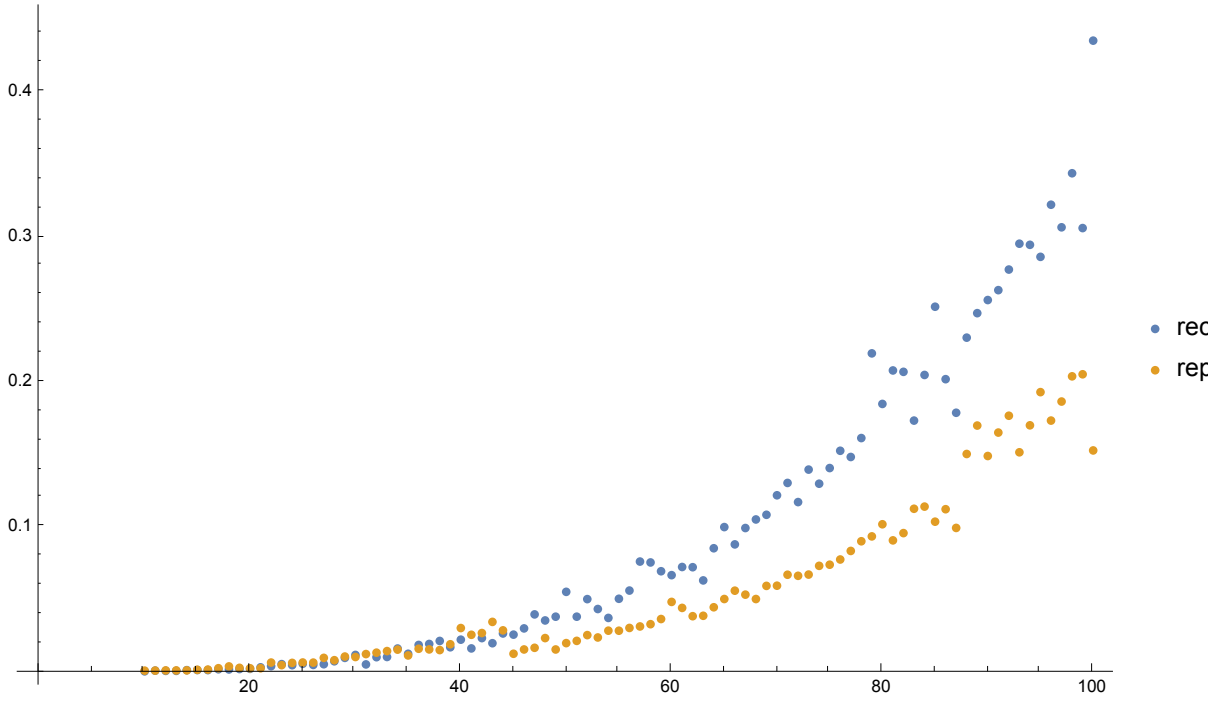
```
{a___, x_, y_, z___} /; x < y → {a, y, x, z}
```

Now for the generation of test data to be compared with the former achieved data of the recursive bubble sort

```
timingBubbleSortReplacement =
  Table[{n, First @ (Timing[RandomInteger[1000, n] //. bubbleReplacementRule])},
   {n, 10, 100}];
```

Now how about the results compared:

```
ListPlot[{timingBubbleSortRule, timingBubbleSortReplacement},
 PlotStyle → PointSize[Medium], ImageSize → Large,
 PlotLegends → {"recursive", "replacement"}]
```

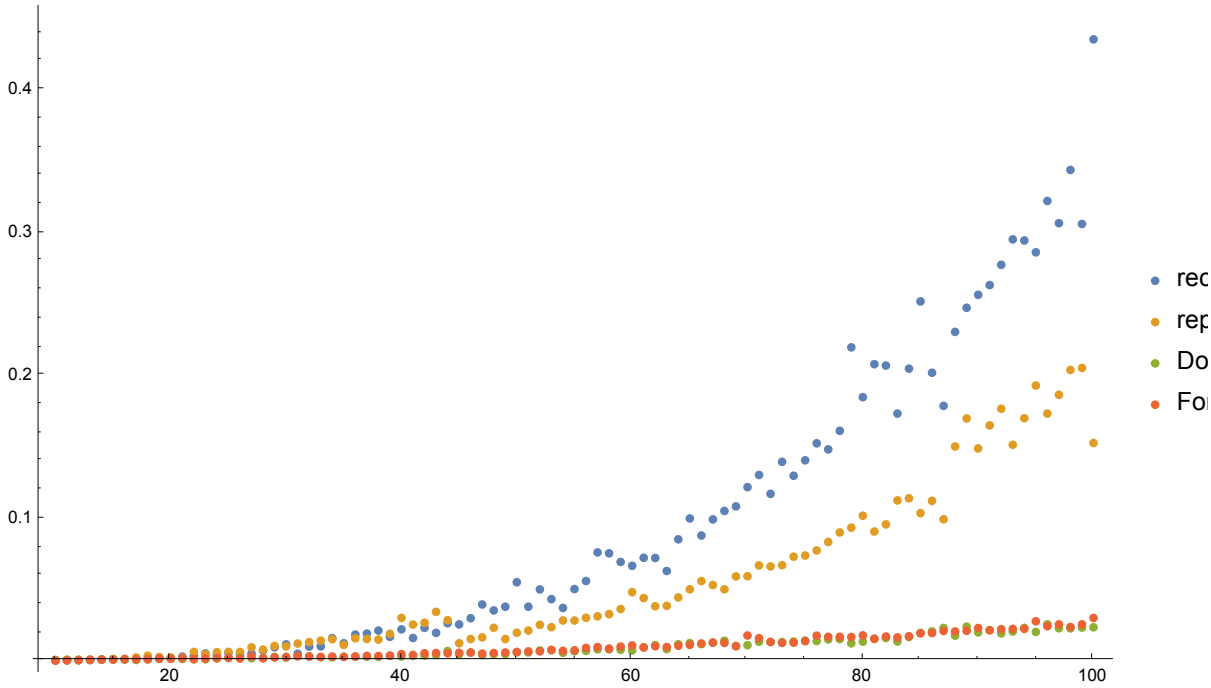


So the approach with replacement rules is by far more effective than the usage of a recursive rule based approach. So putting the parts together and compare (visually) the four implementations we get the following picture (ee recalculate the timings to hae it on the same scale):

```
timingBubbleSortCompare =
  Table[{n, First @ (Timing[bubbleSort @ RandomInteger[1000, n]])}, {n, 10, 100}];

timingBubbleSortForCompare = Table[
   {n, First @ (Timing[bubbleSortFor @ RandomInteger[1000, n]])}, {n, 10, 100}];
```

```
ListPlot[{timingBubbleSortRule, timingBubbleSortReplacement,
  timingBubbleSortCompare, timingBubbleSortForCompare},
 PlotStyle → PointSize[Medium], ImageSize → Large,
 PlotLegends → {"recursive", "replacement", "Do", "For"}, PlotRange → All]
```



Now we have a rather clear picture of bubble sort.

Init

Set the correct environment

```
Switch[SystemInformation["Kernel", "MachineName"],
  "MacBook-Pro-mg",
  SetDirectory[$UserDocumentsDirectory ~~ "/OneDrive/Mathematica"],
  "Michaels-Air", SetDirectory["/Users/mgAir/OneDrive/Mathematica"],
  True, $UserDocumentsDirectory
 ];
```

```
$PlotTheme = "Scientific";
```

We have a lot of results for various sorting algorithms in a dataset, this is now read in:

```
sortingTime = Import["timing.m"];
```

For filling in more data it is convenient to have a function that does exactly this, so we make one. We define a function that appends calculated data on our dataset and sores the result for future reference: