Plain sorting

Input Data size: **2MB** ; Input Data type: ASCII records

Programming language **: Python**

Sorting Algorithm : **Bubble sort**

To analyze the running time, CPU consumption and time spent in each function call, I have used a deterministic python profiler called cProfile. The results are displayed below.

```
         200050007 function calls in 115.286 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  115.286  115.286 <string>:1(<module>)
        1   84.865   84.865  115.259  115.259 bubble.py:12(bubbleSort)
        1    0.014    0.014  115.285  115.285 bubble.py:4(main)
199990000   29.397    0.000   29.397    0.000 {cmp}
        1    0.000    0.000    0.000    0.000 {len}
    20000    0.001    0.000    0.001    0.000 {method 'append' of 'list'
objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
    20000    0.011    0.000    0.011    0.000 {method 'write' of 'file'
objects}
        2    0.000    0.000    0.000    0.000 {open}
    20000    0.997    0.000    0.997    0.000 {range}
```

Bubble sort is a quadratic time in-place sorting algorithm.From the results above, it can be seen that most of calls were made to the cmp function. Most amount of the time is spent in bubbleSort() function as seen from percall, cumtime values.

I know that Bubble sort uses O(1) space, but, just wanted to see how to memory profiling would be. And the results are presented below.

```
Line #    Mem usage    Increment   Line Contents
================================================
     4    8.676 MiB    0.000 MiB   @profile
     5                             def main():
     6    8.680 MiB    0.004 MiB    alist =[]
     7    8.680 MiB    0.000 MiB    swapcnt=0
     8    8.680 MiB    0.000 MiB    f = open('2MB', 'r+b')
     9    8.680 MiB    0.000 MiB    fw = open('smallbpy', 'r+b')
    10   11.738 MiB    3.059 MiB    for line in f:
    11   11.738 MiB    0.000 MiB            alist.append(line)
    12
    13   11.738 MiB    0.000 MiB    def bubbleSort(alist):
    14                                      swapcnt =0
    15                                      for passnum in range(len(alist)-1,0,-1):
    16                                              for i in range(passnum):
    17                                                      if cmp(alist[i],alist[i+1])==1:
    18                                                              temp = alist[i]
    19                                                              alist[i] = alist[i+1]
    20                                                              alist[i+1] = temp
    21                                                              swapcnt=swapcnt+1
    22
    23
    24    8.781 MiB   -2.957 MiB    bubbleSort(alist)
    25    8.840 MiB    0.059 MiB    for l in alist:
    26    8.840 MiB    0.000 MiB            fw.write(str(l))
    27    8.859 MiB    0.020 MiB    print "number of swaps",swapcnt
```

Since bubble sort is an in-place sort algortithm with a space complexity of O(1), not much of memory is utilized in sorting data.

Programming language **: Python** Sorting Algorithm : **Heap Sort**

A simple "time" command on the python script that is running the heap sort gives the following output :

First run:

```
real 0m0.239s
user 0m0.064s
sys  0m0.017s
```

Second run:

```
real 0m0.217s
user 0m0.075s
sys  0m0.032s
```

Third run:

```
real 0m0.077s
user 0m0.061s
sys  0m0.014s
```

Fourth run:

```
real 0m0.074s
user 0m0.060s
sys  0m0.013s
```

Fifth run:

```
real 0m0.077s
user 0m0.061s
sys  0m0.014s
```

Sixth run:

```
real 0m0.079s
user 0m0.064s
sys  0m0.014s
```

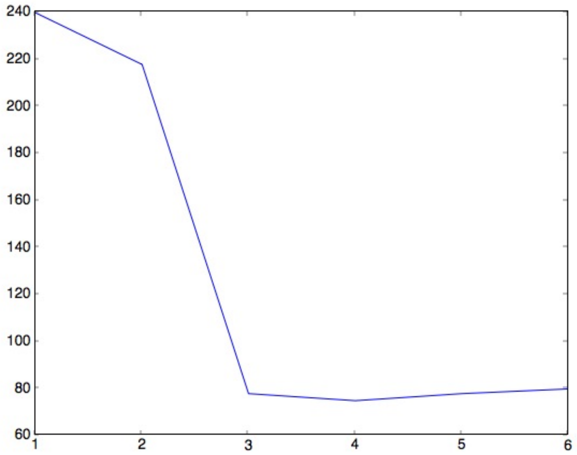


Fig 1

In order to analyze the results based on few runs, I ran the same code many times, the results were displayed above in fig 1.

- It can be observed that the first few runs had a high user time (compared to the user and system time), meaning that they involved a lot of I/O processing than CPU processing.
- After a few runs, the user time has come closer and reduced quite a lot, this can be attributed to bringing the data to memory/ cache resulting in reduced overall running time.

To analyze the results more deeply, I ran a memory profiler on the same program and the results are listed below.

```
          60007 function calls in 0.062 seconds

        Ordered by: standard name

        ncalls  tottime  percall  cumtime  percall filename:lineno(function)
             1    0.001    0.001    0.062    0.062 heap.py:1(<module>)
             1    0.017    0.017    0.060    0.060 heap.py:3(main)
             1    0.001    0.001    0.001    0.001 heapq.py:31(<module>)
             1    0.003    0.003    0.003    0.003 {_heapq.heapify}
         20000    0.028    0.000    0.028    0.000 {_heapq.heappop}
         20000    0.002    0.000    0.002    0.000 {method 'append' of 'list'
     objects}
             1    0.000    0.000    0.000    0.000 {method 'disable' of
     '_lsprof.Profiler' objects}
         20000    0.011    0.000    0.011    0.000 {method 'write' of 'file'
     objects}
             2    0.000    0.000    0.000    0.000 {open}
```

As it can be seen from the above profiling results, most of the time is being spent in running heappop() function. The heappop() function basically pops and returns the smallest item from the *heap*, maintaining the heap invariant.

Heap sort is also uses O(1) space, I ran memory profiling on the script and the results are reported below.

```
Line #    Mem usage    Increment   Line Contents
================================================
     2    8.750 MiB    0.000 MiB   @profile
     3                             def main():
     4    8.754 MiB    0.004 MiB     heap =[]
     5    8.754 MiB    0.000 MiB     f = open('2MB', 'r+b')
     6    8.754 MiB    0.000 MiB     fw = open('smallhpy', 'r+b')
     7   11.664 MiB    2.910 MiB     for line in f:
     8   11.664 MiB    0.000 MiB       heap.append(line)
     9
    10   11.664 MiB    0.000 MiB     heapq.heapify(heap)
    11
    12   11.734 MiB    0.070 MiB     while heap:
    13   11.734 MiB    0.000 MiB       fw.write(heapq.heappop(heap))
```

Though not much of memory is consumed, major portions of it is consumed at the filw write operation and the heapify function.

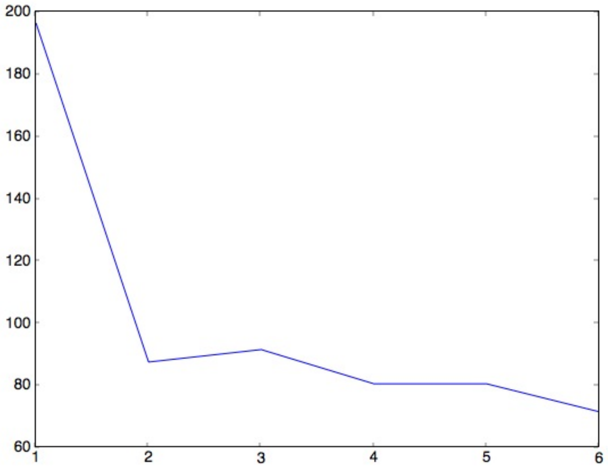
Programming language **: Go** Sorting Algorithm : **Bubble Sort**

A simple "time" command on the go program that is running the bubble sort gives the following output :

```
First run:
real	0m8.490s
user	0m6.353s
sys	0m0.196s
Second run
real	0m6.646s
user	0m6.260s
sys	0m0.087s
Third run:
real	0m6.797s
user	0m6.311s
sys	0m0.091s
Fourth run:
real	0m6.645s
user	0m6.302s
sys	0m0.080s
fifth run:
real	0m6.653s
user	0m6.239s
sys	0m0.080s

Sixth run:
real	0m6.492s
user	0m6.271s
sys	0m0.071s
```

Fig 2



In order to analyze the results based on few runs, I ran the same code many times, the results were displayed above in fig 2. I could infer that Go language has finished the bubble sort on the same input data faster than python did. Also, there is a decline in the kernel CPU time as the number of attempts increased. This is a very good thing to notice as there are very less number of system calls inside the kernel.

Programming language **: Go** Sorting Algorithm : **Heap Sort**

A simple "time" command on the go sprogram that is running the heap sort gives the following output :

First run :

```
real 0m0.210s
user 0m0.048s
sys  0m0.025s
```

Second run:

```
real 0m0.054s
user 0m0.039s
sys  0m0.014s
```

Third run:

```
real 0m0.051s
user 0m0.037s
sys  0m0.013s
```

We can observe the decrease in user time as the number of attempts increase.

Input Data size: **10MB** ; Input Data type: ASCII records

Programming language **: Python** Sorting Algorithm : **Bubble sort**

This operation took the most time out of all the tests I ran. Results are as follows.

```
       5000250009 function calls in 3316.641 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.009    0.009 3316.641 3316.641 bubble.py:1(<module>)
        1 2473.199 2473.199 3316.311 3316.311 bubble.py:13(bubbleSort)
        1    0.234    0.234 3316.633 3316.633 bubble.py:5(main)
        1    0.000    0.000    0.000    0.000 cProfile.py:5(<module>)
        1    0.000    0.000    0.000    0.000 cProfile.py:66(Profile)
4999950000  816.470    0.000  816.470    0.000 {cmp}
        1    0.000    0.000    0.000    0.000 {len}
   100000    0.010    0.000    0.010    0.000 {method 'append' of 'list'
objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
   100000    0.078    0.000    0.078    0.000 {method 'write' of 'file'
objects}
        2    0.000    0.000    0.000    0.000 {open}
   100000   26.643    0.000   26.643    0.000 {range}
```

```
real 55m16.917s
user 55m13.392s
sys  0m2.387s
```

Programming language **: Python** Sorting Algorithm : **Heap sort**

After running a heap sort on 10MB data, the CPU utilization snapshot looks like this

```
       300007 function calls in 0.609 seconds

        Ordered by: standard name

        ncalls  tottime  percall  cumtime  percall filename:lineno(function)
             1    0.115    0.115    0.609    0.609 heap.py:1(<module>)
             1    0.105    0.105    0.436    0.436 heap.py:3(main)
             1    0.058    0.058    0.058    0.058 heapq.py:31(<module>)
             1    0.021    0.021    0.021    0.021 {_heapq.heapify}
        100000    0.212    0.000    0.212    0.000 {_heapq.heappop}
        100000    0.007    0.000    0.007    0.000 {method 'append' of 'list'
     objects}
             1    0.000    0.000    0.000    0.000 {method 'disable' of
     '_lsprof.Profiler' objects}
        100000    0.090    0.000    0.090    0.000 {method 'write' of 'file'
     objects}
             2    0.000    0.000    0.000    0.000 {open}

     real 0m1.387s
     user 0m0.401s
     sys   0m0.068s
```

Majority of the time here Is being spent in the heappop operation just like in the previous case.

Memory utilization of python – Heapsort looks like this :

```
     Line #    Mem usage    Increment   Line Contents
     ================================================
          2    8.465 MiB    0.000 MiB   @profile
          3                             def main():
          4    8.469 MiB    0.004 MiB     heap =[]
          5    8.469 MiB    0.000 MiB     f = open('10MB', 'r+b')
          6    8.469 MiB    0.000 MiB     fw = open('bighpy', 'r+b')
          7   22.793 MiB   14.324 MiB     for line in f:
          8   22.793 MiB    0.000 MiB       heap.append(line)
          9
         10   23.488 MiB    0.695 MiB     heapq.heapify(heap)
         11
         12   23.488 MiB    0.000 MiB     while heap:
         13   23.488 MiB    0.000 MiB       fw.write(heapq.heappop(heap))
```

Through there isn't a significant usage of memory here, we can observe that the heapify() function is using twice the memory than before.

Programming language **: Go** Sorting Algorithm : **Bubble Sort**

Though the script ran for a long time, it is still a winner compared to the python version of the same algorithm.

1)

```
     real 4m8.040s
     user 4m6.883s
     sys   0m0.391s
```

2)

```
real 4m9.628s
user 4m8.508s
sys   0m0.421s
```

Programming language **: Go** Sorting Algorithm : **Heap Sort**

I ran the go script for heap sort on 10 MB file and the results are as follows:

First run:

```
real 0m2.615s
user 0m0.521s
sys  0m0.134s
```

Second run:

```
real 0m1.427s
user 0m0.530s
sys  0m0.121s
```

Third run:

```
real 0m0.676s
user 0m0.511s
sys  0m0.084s
```

Fourth run:

```
real 0m0.741s
user 0m0.521s
sys  0m0.083s
```

Fifth run:

```
real 0m0.612s
user 0m0.518s
sys  0m0.081s
```

Sixth run:

```
real 0m0.618s
user 0m0.530s
sys  0m0.078s
```

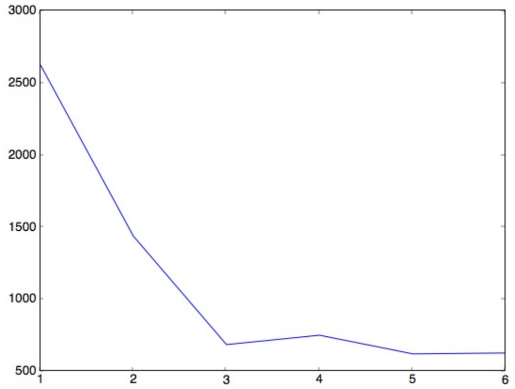
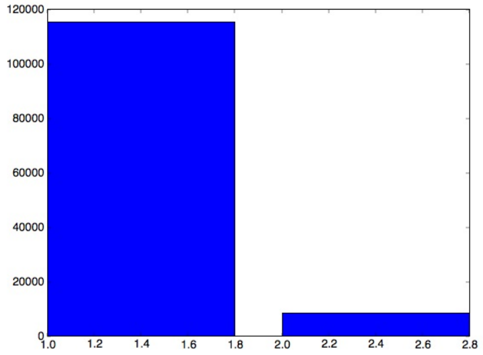
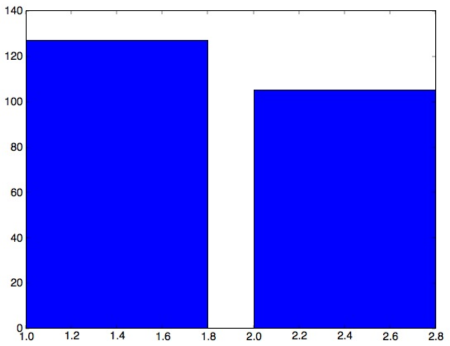


Fig 4

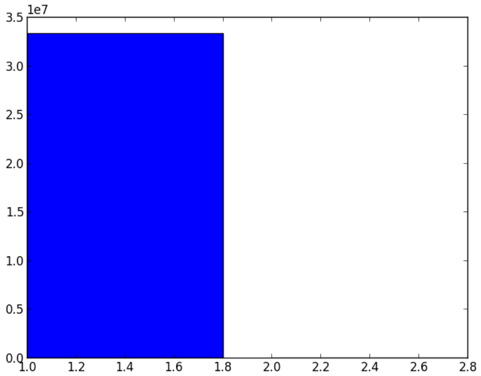
Analysis



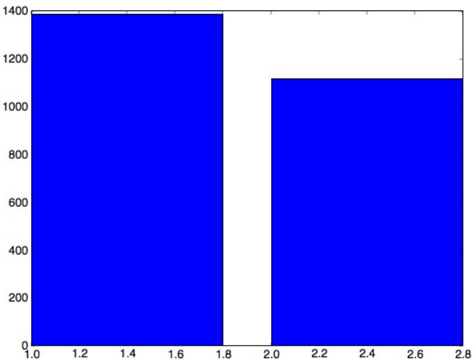
Comparing Bubble sort performance
(Metric : Highest CPU time (ms))
Python on left, Go on right (2MB)



Comparing Heap sort performance
(Metric : Average CPU time (ms))
Python on left, Go on right (2MB)



Comparing Bubble sort performance
(Metric : Highest CPU time (ms))
Python on left Go on right (10MB)



Computing Heap sort performance
Metric : Highest CPU time (ms))
Python on left Go on right (10MB)

Discussion

- Overall, Go looks like a better choice to perform sorting on small / large data.
- I have observed that the kernel call time of sorting performed using Go language is really low.
- On the other hand, Python catches up well against Go in Heapsort.
- We might want to consider other options in python like “stackless python” before migrating.