Sorting Algorithms

June 15, 2020

1 Sorting Algorithms

Speed comparison of 4 sorting methods: BS, HS, CS, ShS for the array of integers randomly generated according to the uniform probability distribution.

[24]:
```
import random
import time
import pandas as pd
import matplotlib.pyplot as plt
import sys
import numpy as np
sys.setrecursionlimit(10**7)
```

1.1 Data Types

[25]:
```
def Randomlst(length):
    return [random.randint(0,10*length) for i in range(length)]
def Constant(length):
    return [0 for i in range(length)]
def Increasing(length):
    return  [i for i in range(100,length,10)]
def Decreasing(length):
    return Increasing(length)[::-1]
def Ashape(length):
    l = [i for i in range(length//2) if i%2 ==1]
    p = [i for i in range(length//2,0,-1) if i%2 ==0]
    return  l+p
def Vshape(length):
    l = [i for i in range(length//2,0,-1) if i%2 ==1]
    p = [i for i in range(length//2) if i%2 ==0]
    return  l+p
```

Function that checks if algorithms are implemented correctly

[26]:
```
def check(func):
    lst = Randomlst(1000)
```

```
    if type(func(lst)) is list:
        if sorted(lst) == func(lst):
            return ("Algorithm is working")
        return "Algorithm is not working"
    if sorted(lst) == func(lst)[0]:
        return ("Algorithm is working")
    return "Algorithm is not working"
```

1.2 Bubble Sort

[27]:
```
def BS(lst):
    time1= time.time()
    for i in range(len(lst)):
        for k in range(len(lst)-i-1):
            if lst[k] > lst[k+1]:
                lst[k],lst[k+1] = lst[k+1],lst[k]
    return lst,time.time() - time1
```

[28]:
```
print(check(BS))
```

```
Algorithm is working
```

1.3 Heap Sort

[29]:
```
def heapify(lst, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and lst[i] < lst[l]:
        largest = l
    if r < n and lst[largest] < lst[r]:
        largest = r
    if largest != i:
        lst[i],lst[largest] = lst[largest],lst[i]
        heapify(lst, n, largest)
def HS(lst):
    time1 = time.time()
    n = len(lst)
    for i in range(n, -1, -1):
        heapify(lst, n, i)
    for i in range(n-1, 0, -1):
        lst[i], lst[0] = lst[0], lst[i]
        heapify(lst, i, 0)
    return lst,time.time() - time1
```

[30]:
```
print(check(HS))
```

```
Algorithm is working
```

1.4 Counting Sort

[31]:
```
def CS(lst):
    time1 = time.time()
    m = max(lst)
    counter = [0]*(m+1)
    output = [0]*len(lst)

    for i in range(len(lst)):
        counter[lst[i]]+=1

    for i in range(len(counter)-1):
        counter[i+1] = counter[i+1] + counter[i]

    for i in range(len(lst)):
        output[counter[lst[i]]-1] = lst[i]
        counter[lst[i]]-=1

    for i in range(len(lst)):
        lst[i] = output[i]

    return lst,time.time() - time1
```

[32]:
```
print(check(CS))
```

```
Algorithm is working
```

1.5 Shell Sort

[33]:
```
def SHS(lst):
    time1 = time.time()
    n = len(lst)//2
    while n >0:
        for i in range(n,len(lst)):
            curr = lst[i]
            j = i
            while j >= n and lst[j-n] > curr:
                lst[j] = lst[j - n]
                j -= n
            lst[j] = curr
        n //=2
```

```
    return lst,time.time() - time1
```

[34]:
```
print(check(SHS))
```

```
Algorithm is working
```

1.6 Creating some useful Dataframe

[35]:
```
df = [[] for i in range(100,2100,100) ]
algorithms =[BS,HS,CS,SHS]
n=0

for i in range(100,2100,100):
    for k in algorithms:
        normalization = []
        for m in range(10):
            normalization.append(k(Randomlst(i))[1])
        df[n].append(np.mean(normalization))
    n +=1
df = pd.DataFrame(df,columns = ['Bubble Sort','Heap Sort','Counting␣
↪Sort','Shell Sort'],\
                   index=[i for i in range(100,2100,100)] )
df.index.name = 'length '
df2 =df.loc[:, df.columns != 'Bubble Sort']
df
```

[35]:

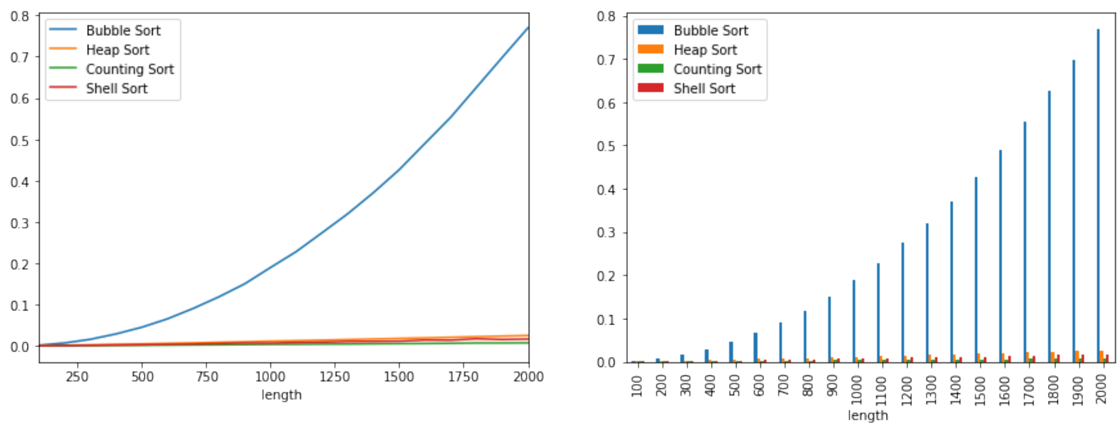
length	Bubble Sort	Heap Sort	Counting Sort	Shell Sort
100	0.001892	0.000798	0.000390	0.000412
200	0.007227	0.001725	0.000740	0.001018
300	0.016053	0.002765	0.001102	0.001584
400	0.029093	0.003888	0.001464	0.002419
500	0.045220	0.005046	0.001847	0.002908
600	0.065601	0.006265	0.002232	0.003866
700	0.090792	0.007559	0.002651	0.004654
800	0.118934	0.008786	0.003002	0.006047
900	0.150336	0.010069	0.003362	0.007405
1000	0.189700	0.011745	0.003890	0.007094
1100	0.228286	0.012749	0.004116	0.008473
1200	0.273743	0.014113	0.004542	0.009376
1300	0.319830	0.015422	0.004872	0.010896
1400	0.371000	0.016912	0.005314	0.011064
1500	0.426428	0.018225	0.005734	0.011704
1600	0.490065	0.019506	0.006020	0.014400
1700	0.553377	0.020994	0.006490	0.014056
1800	0.625770	0.022426	0.007286	0.017415

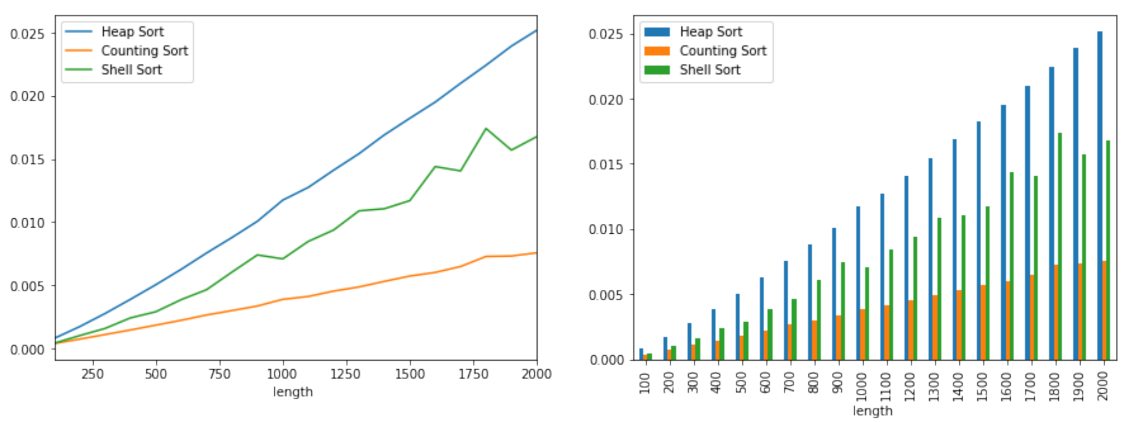
1900	0.698092	0.023939	0.007321	0.015704
2000	0.769729	0.025196	0.007574	0.016773

1.7 Plots

[36]:
```
for i in [df,df2]:
    fig, axes = plt.subplots(nrows =1, ncols =2, figsize=(15,5))
    i.plot(kind = 'line', ax = axes[0])
    i.plot(kind = 'bar', ax = axes[1])

    plt.show()
```





1.8 Conclusions

To simplify drawing conclusions, I presented two pairs of plots. It can be easily observed that in the first pair of charts Bubble Sort's time is greatly larger in comparison to other algorithms. In the second pair of graphs, worth noticing is the lowest time of execution which belongs to Counting Sort with its complexity equal to O(n). Both Heap Sort and Shell Sort presents similar scores with the marginal advantage of Shell Sort. Either Heap Sort or Shell Sort being better than Counting Sort in terms of memory allocation, which unlike them sorts not in place. Although Heap Sort works seemingly faster in comparison to Shell Sort, Shell Sort has lower complexity than Heap Sort. Aforementioned effect is caused due to the fact that n^1.25 is smaller than n*log(n) for small numbers.* **Counting Sort** time: O(n+k) memory: O(n+k) where k is equal to 10*n* **Bubble Sort** time: O(n^2) memory: O(n) * **Heap Sort** time: O(nlogn) memory: O(n) * **Shell Sort** time: O(n^1.25) memory: O(n)

2 Exercise 2

Effectiveness comparison of 3 sorting methods: QS with middle selected pivot, HS, MS. Examined for the following data types:

- random (uniform distribution)
- constant value (e.g.equal to 0)
- increasing order (step equal to 1)
- descending order (step equal to 1)
- ascending-descending order (A shape – increase odd numbers - decrease even)
- descending-ascending order (V -shape – decrease odd numbers - increase even)

2.1 Quick Sort

[37]:
```
def _QS(array, start, end):
    pivot = array[int(np.floor(start + (end - start) / 2))]
    a = start - 1
    b = end + 1
    while True:
        a += 1
        while array[a] < pivot:
            a += 1
        b -= 1
        while array[b] > pivot:
            b -= 1
        if a >= b:
            return b
        array[a], array[b] = array[b], array[a]


def innerQS(array,start,end):
    if start < end:
```

```
        pivot = _QS(array, start, end)
        innerQS(array, start, pivot)
        innerQS(array, pivot + 1, end)
    return array
def QS(array):
    return innerQS(array,0,len(array)-1)
```

[38]:
```
print(check(QS))
# time1 = time.time()
# QS(Randomlst(1000000))
# print(time.time()-time1)
```

```
Algorithm is working
```

2.2 Merge Sort

[39]:
```
def MS(lst):
    l = len(lst)
    if l <= 1:
        return lst
    result = []
    y = MS(lst[:(l // 2)])
    z = MS(lst[(l // 2):])
    i = 0
    j = 0
    while i < len(y) and j < len(z):
        if y[i] > z[j]:
            result.append(z[j])
            j += 1
        else:
            result.append(y[i])
            i += 1
    result += y[i:]
    result += z[j:]
    return result
```

[40]:
```
print(check(MS))
```

```
Algorithm is working
```

2.3 Creating some useful DataFrames

[41]:
```
def creatingDataFrame(data,names):
    df = [[] for i in range(6)]
    algth = [QS, HS,MS]
    n = 0
    for idx,k in enumerate(algth):
        for datatype in data:
            for i in range(100,2000,100):
                normalization = []
                for v in range(10):
                    time1 = time.time()
                    k(datatype(i))
                    time2 = time.time() - time1
                    normalization.append(time2)

                df[n].append(np.mean(normalization))
            n +=1
    df = list(zip(*df))
    arr = [['Quick Sort','Quick Sort','Heap Sort','Heap Sort','Merge␣
↪Sort','Merge Sort'],names*3]
    tuples = list(zip(*arr))
    indexes = pd.MultiIndex.from_tuples(tuples, names=['Algorithms', 'Data␣
↪Type'])
    df2 = pd.DataFrame(df, columns = indexes, index = [i for i in␣
↪range(100,2000,100)])
    return df2
```

[42]:
```
scores1 = creatingDataFrame([Randomlst,Constant],['Random', 'Constant'])
scores1
```

[42]:

Algorithms	Quick Sort		Heap Sort		Merge Sort	
Data Type	Random	Constant	Random	Constant	Random	Constant
100	0.000933	0.000624	0.001019	0.000208	0.000855	0.000418
200	0.001967	0.001314	0.002267	0.000413	0.001897	0.000898
300	0.002905	0.002021	0.003639	0.000619	0.002957	0.001414
400	0.003809	0.002767	0.005016	0.000828	0.004010	0.001926
500	0.005088	0.003565	0.006607	0.001035	0.005228	0.002487
600	0.006007	0.004281	0.008040	0.001249	0.006319	0.003017
700	0.007120	0.005401	0.009673	0.001463	0.007426	0.003581
800	0.008022	0.005847	0.011139	0.001675	0.008557	0.004146
900	0.009217	0.006650	0.012899	0.001875	0.009995	0.004739
1000	0.010419	0.007506	0.014498	0.002090	0.011205	0.005369
1100	0.011341	0.008328	0.016035	0.002294	0.012445	0.005935
1200	0.012358	0.009043	0.017593	0.002497	0.013528	0.006639
1300	0.013362	0.009846	0.019239	0.002705	0.014748	0.007083
1400	0.014294	0.010696	0.020899	0.002914	0.015957	0.007674

1500	0.015410	0.011423	0.022422	0.003126	0.017085	0.008305
1600	0.016464	0.012244	0.024107	0.003325	0.018313	0.008877
1700	0.018048	0.013081	0.026249	0.003531	0.020048	0.009481
1800	0.019242	0.013996	0.027960	0.003751	0.021270	0.010217
1900	0.020101	0.014829	0.029595	0.003962	0.022531	0.010816

[43]:
```
scores2 = creatingDataFrame([Increasing, Decreasing],['Increasing',␣
↪'Decreasing'])
scores2
```

[43]:

Algorithms	Quick Sort		Heap Sort		Merge Sort	
Data Type	Increasing	Decreasing	Increasing	Decreasing	Increasing	Decreasing
100	0.000002	0.000003	0.000004	0.000005	0.000002	0.000003
200	0.000044	0.000047	0.000044	0.000035	0.000034	0.000045
300	0.000093	0.000097	0.000108	0.000091	0.000072	0.000078
400	0.000142	0.000158	0.000178	0.000150	0.000109	0.000118
500	0.000222	0.000198	0.000259	0.000212	0.000154	0.000167
600	0.000239	0.000259	0.000343	0.000282	0.000197	0.000208
700	0.000300	0.000314	0.000425	0.000355	0.000286	0.000252
800	0.000342	0.000368	0.000519	0.000436	0.000402	0.000301
900	0.000397	0.000416	0.000612	0.000533	0.000415	0.000350
1000	0.000458	0.000472	0.000711	0.000598	0.000379	0.000408
1100	0.000505	0.000525	0.000804	0.000688	0.000426	0.000454
1200	0.000564	0.000587	0.000909	0.000752	0.000474	0.000507
1300	0.000690	0.000649	0.000982	0.000835	0.000526	0.000547
1400	0.000692	0.000696	0.001093	0.000919	0.000584	0.000593
1500	0.000717	0.000784	0.001206	0.001187	0.000634	0.000654
1600	0.000812	0.000798	0.001311	0.001320	0.000661	0.000716
1700	0.000829	0.000846	0.001476	0.001301	0.000719	0.000772
1800	0.000880	0.000896	0.001534	0.001569	0.000760	0.000825
1900	0.000951	0.000954	0.001703	0.001561	0.000811	0.000880

[44]:
```
scores3 = creatingDataFrame([Ashape,Vshape],['Ashape','Vshape'])
scores3
```

[44]:

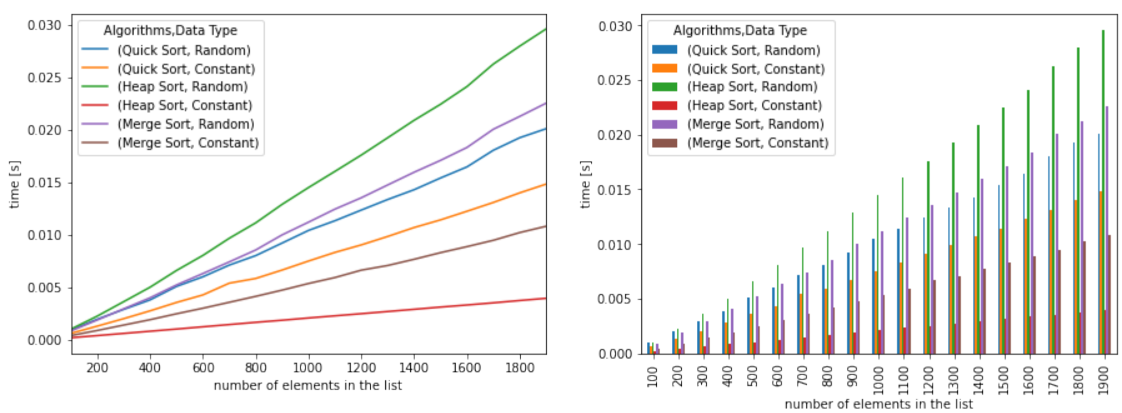
Algorithms	Quick Sort		Heap Sort		Merge Sort	
Data Type	Ashape	Vshape	Ashape	Vshape	Ashape	Vshape
100	0.000362	0.000332	0.000340	0.000307	0.000231	0.000229
200	0.000928	0.000728	0.000768	0.000720	0.000494	0.000494
300	0.001760	0.001166	0.001254	0.001172	0.000769	0.000770
400	0.002667	0.001606	0.001775	0.001656	0.001063	0.001063
500	0.003789	0.002125	0.002323	0.002128	0.001334	0.001363
600	0.005105	0.002599	0.002870	0.002667	0.001645	0.001666
700	0.006421	0.003080	0.003468	0.003235	0.001944	0.001977
800	0.008245	0.003639	0.004041	0.003733	0.002254	0.002293
900	0.010079	0.004363	0.004662	0.004320	0.002561	0.002563
1000	0.012154	0.005022	0.005260	0.004856	0.002863	0.002854

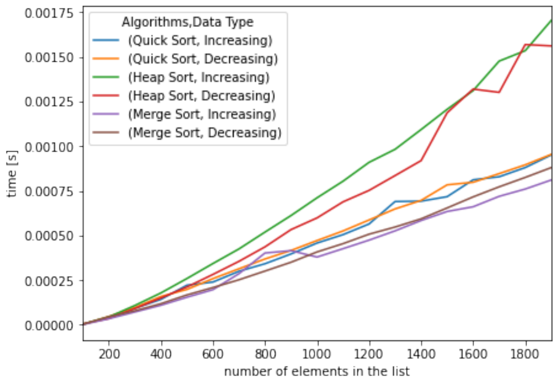
1100	0.014611	0.005422	0.005863	0.005420	0.003193	0.003193
1200	0.017037	0.005776	0.006480	0.006034	0.003518	0.003515
1300	0.019644	0.006396	0.007132	0.006651	0.003839	0.003859
1400	0.022597	0.007298	0.007757	0.007248	0.004166	0.004173
1500	0.025605	0.007749	0.008400	0.007791	0.004493	0.004553
1600	0.029264	0.008409	0.009073	0.008403	0.004819	0.004813
1700	0.032797	0.008771	0.009814	0.009111	0.005164	0.005175
1800	0.037333	0.009350	0.010452	0.009644	0.005570	0.005509
1900	0.039980	0.010353	0.011141	0.010270	0.005810	0.005774

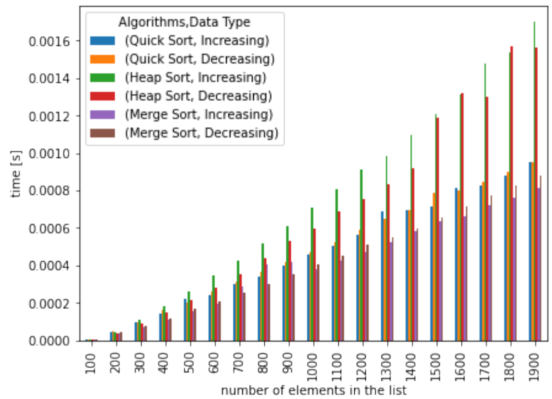
[ ]:
```
jupyter nbconvert --to htmlsorting.ipynb
```

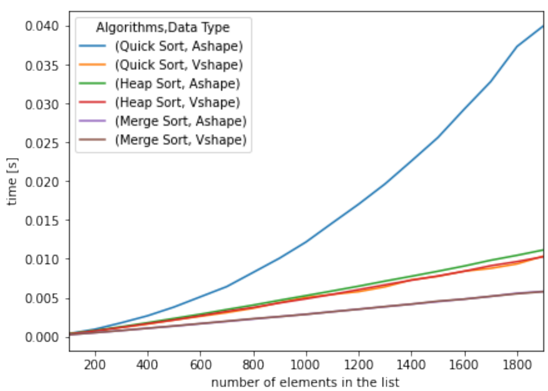
2.4 Plots

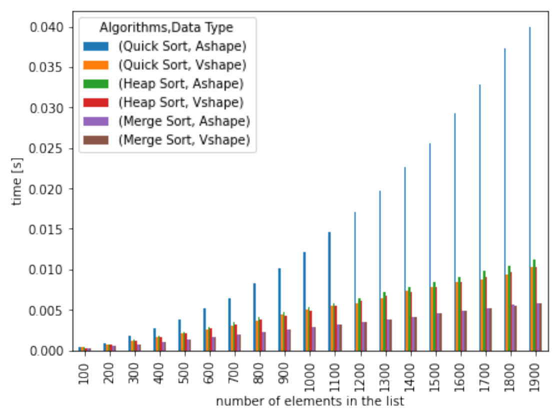
[45]:
```
scores = [scores1,scores2,scores3]
for i in scores:
    fig, axes = plt.subplots(nrows =1, ncols =2, figsize=(15,5))
    l = i.plot(kind = 'line', ax = axes[0])
    p = i.plot(kind = 'bar', ax = axes[1])
    l.set(xlabel="number of elements in the list", ylabel="time [s]")
    p.set(xlabel="number of elements in the list", ylabel="time [s]")
    plt.show()
```











2.5 Conclusions

While sorting arrays either filled with a constant value or in so-called ”V-Shape” quick sort execution time rockets significantly. It is caused by producing n-1 splits in which all elements are moved to one side of the pivot which contributes to the complexity of the algorithm equal to O(n^2), its worst-case scenario. The best efficiency of Quick Sort algorithm can be seen while operating on already sorted arrays, elements are already in the right position so there is no need to swap elements. Worth noticing is the impact of choosing the median as a pivot, which ensures one to produce a sorting algorithm with O(n log n) running time. If one were to choose a different pivot, performance for already sorted data would dramatically decrease.

- **Quick Sort** time: O(nlog(n)) memory: O(n)
- **Heap Sort** time: O(nlog(n)) memory: O(n)
- **Merge Sort** time: O(nlog(n)) memory: O(n)

Project created by:

Maciej Filanowicz