Program 4: Benchmarking Heaps

Mikhail Filippov

In this assigment, we will implement **three** approaches to heapifying an array. Specifically, we are trying to build a max heap. The three approaches will be:

1. Insertion Method: Insert elements one at a time into the heap
2. Bubble down until value is bigger than both children (we'll call this "Bubble Method 1")
3. Bubble down all the way, then bubble up until value is smaller than parent (we'll call this "Bubble Method 2")

With each one, we will benchmark on **sorted**, **reverse sorted**, and **random** arrays to test the `heapify` method and analyze using regression techniques to estimate asymptotic time complexity for each case.

Planned Approach

For our three cases to benchmark, we will be using **sorted**, **reverse sorted**, and **randomly shuffled** lists to heapify with each approach. Generating each list won't be too much of a problem, and we will increase the size of each list linearly to look at the graphs and slopes generated by the dependant time output by the benchmarks.

Expected results and predictions:

1. Insertion method: We know that an insertion takes $O(log(n))$ time, so to insert n number of elements, we should get a time complexity of $O(n \times log(n))$. Since we are inserting at the end of the heap and bubbling up, the expected best case should be a **reverse sorted** list, as we will be adding the biggest element at `A[0]` first, followed by all elements at `A[i]` with $i > 0$, with each element `A[i+1]` $<$ `A[i]`. This allows for all smaller elements to be added to the bottom and not needed to be bubbled up. This, should, reduce our best case runtime to $O(n)$, as we are adding elements in costant time n number of times now.
2. Bubble Method 1: We know the recurrence relation for this method is $T(n) = 2T(\frac{n}{2}) + \Theta(log(n))$. In the best case though, **reverse sorted** again, the $\Theta(log(n))$ part is insignificant. This reduces our recurrence to $T(n) = 2T(\frac{n}{2}) + \Theta(1)$ since we still need to call `BubbleDown`. This is still reduced to $O(n)$, but it should theoretically run faster than any other case.
3. Bubble Method 2: This approach's best case should be the **sorted** list. This happens since no matter what, we bubble each value down **all** the way, and bubble back up *until the value is smaller than parent*. In a sorted list, this ensures that all of the smaller values

will be bubbled down to the end and won't have to be bubble back up. The asymptotic running time for this should be $O(n)$.

Heapification Approaches

1. Insertion

```
def insertion_heapify(A):
    """
    Heapify using insertion method and adding elements one by one
    """
    heap = []
    for i in range(len(A)):
        insert(heap, A[i])
    A[:] = heap
```

2. Heapification Function

```
def heapify(A, bubble_down=bubble_method_1):
    """
    Heapify using the passed in bubble_down method and build the heap
buttom up
    """
    heapify_h(A, 0, bubble_down)

def heapify_h(A, i, bubble_down):
    """
    Heapify wrapper calling helper
    """
    if i < len(A)//2:
        heapify_h(A, left(i), bubble_down)
        heapify_h(A, right(i), bubble_down)
        bubble_down(A, i)
```

2a. Bubble Method 1

```
def bubble_method_1(A, i):
    """
    Bubble down until value is bigger than both children (we'll call this
"Bubble Method 1")
    Choose the bigger child and swap with it while comparing with itself,
then repeat until no more swaps are needed
    """
    n = len(A)
    l = left(i)
    r = right(i)
    while (l < n and A[i] < A[l]) or (r < n and A[i] < A[r]):
        if l < n and r < n:
            # left
            if A[l] > A[r]:
                temp = A[l]
```

```
                A[l] = A[i]
                A[i] = temp
                i = l
            # right
            else:
                temp = A[r]
                A[r] = A[i]
                A[i] = temp
                i = r
        elif l < n:
            # left
            temp = A[l]
            A[l] = A[i]
            A[i] = temp
            i = l
        elif r < n:
            # right
            temp = A[r]
            A[r] = A[i]
            A[i] = temp
            i = r
        l = left(i)
        r = right(i)
```

2b. Bubble Method 2

```
def bubble_method_2(A, i):
    """
    Bubble down all the way, then bubble up until value is smaller than
parent
    Choose the bigger child and swap with it all the way down, then go
back up until smaller than parent
    """
    n = len(A)
    l = left(i)
    r = right(i)
    while l < n:
        if r < n:
            # left
            if A[l] > A[r]:
                temp = A[l]
                A[l] = A[i]
                A[i] = temp
                i = l
            # right
            else:
                temp = A[r]
                A[r] = A[i]
                A[i] = temp
                i = r
        else:
            # left
```

```
            temp = A[l]
            A[l] = A[i]
            A[i] = temp
            i = l
        l = left(i)
        r = right(i)

    # bubble up
    parent_i = parent(i)
    while i > 0 and A[parent_i] < A[i]:
        temp = A[parent_i]
        A[parent_i] = A[i]
        A[i] = temp
        i = parent_i
        parent_i = parent(i)
```

Note: The appendix will contain the full source code implementations for the three approaches and their respective descriptions, as well as any helper methods not described above.

Benchmarking

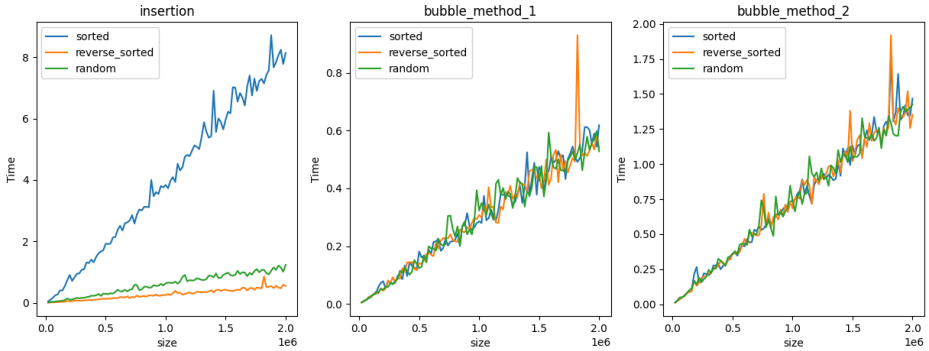
In [104…

```
import dill
import pandas as pd
df = None
with open('heapify_benchmark.db', 'rb') as f:
    df = dill.load(f)
    df = df.astype('float32')
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

In order to effectively benchmark, a list of **100 size values** was used, ranging `20,000` to `2,000,000` in steps of `20,000` to keep linearity in the graph.

The results show that the `insertion` method on a sorted list performed the **worst** out of *any* benchmark by far, although it still appears linear on the graph. The best case for `insertion` was reverse sorted. For the other two methods, `bubble method 1` and `bubble method 2`, the results were consistent accross all cases, with `bubble method 1` being the best method overall for all cases except the reverse sorted list, where it performed the same as `insertion`. The graphs along with their tables representing these relationships can be seen below:

Functions Graphed with Cases



The table is listed below:

In [105…

```
df
```

Out[105...

method	size	sorted	reverse_sorted	random
insertion	**20000**	0.045537	0.004663	0.008509
	40000	0.098389	0.009318	0.021394
	60000	0.163001	0.011982	0.032811
	80000	0.243311	0.019163	0.037184
	100000	0.274680	0.019555	0.051603
	120000	0.401513	0.028707	0.056209
	140000	0.405875	0.027991	0.069824
	160000	0.547294	0.035418	0.087427
	180000	0.737126	0.056284	0.139329
	200000	0.904769	0.043569	0.099155
	220000	0.712207	0.051891	0.103179
	240000	0.825174	0.069393	0.131554
	260000	0.944627	0.074699	0.158704
	280000	0.956449	0.065388	0.144471
	300000	1.076029	0.071435	0.160059
	320000	1.091567	0.076153	0.162478
	340000	1.313120	0.081850	0.183049
	360000	1.294574	0.091096	0.194974
	380000	1.407758	0.080315	0.229591
	400000	1.316853	0.094026	0.225884
	420000	1.491036	0.101531	0.241709
	440000	1.607520	0.108967	0.284244
	460000	1.667969	0.123632	0.224216
	480000	1.710639	0.133435	0.292578
	500000	1.922495	0.124499	0.292993
	520000	1.913166	0.137872	0.279481
	540000	1.926768	0.131448	0.312205
	560000	2.123789	0.152824	0.294815
	580000	2.140429	0.150175	0.354657

method	size	sorted	reverse_sorted	random
	600000	2.374345	0.145777	0.425456
	620000	2.508260	0.188046	0.374832
	640000	2.353572	0.181752	0.370012
	660000	2.586131	0.171461	0.421121
	680000	2.616957	0.208490	0.369166
	700000	2.679156	0.154695	0.422651
	720000	2.852380	0.192123	0.441009
	740000	2.578773	0.175787	0.583975
	760000	2.865786	0.236217	0.575889
	780000	3.039124	0.191864	0.410745
	800000	3.003829	0.209003	0.434902
	820000	3.121313	0.217995	0.518600
	840000	3.121405	0.199812	0.525704
	860000	3.112754	0.241530	0.490223
	880000	3.997786	0.241664	0.500244
	900000	3.471544	0.227389	0.571238
	920000	3.603910	0.268457	0.550536
	940000	3.539551	0.229599	0.553197
	960000	3.801804	0.231916	0.616049
	980000	3.756247	0.253934	0.563885
	1000000	3.836056	0.249507	0.633780
	1020000	3.729624	0.280909	0.655106
	1040000	3.973086	0.252417	0.647881
	1060000	4.092168	0.312539	0.642326
	1080000	3.932088	0.387307	0.685880
	1100000	4.525342	0.314593	0.622706
	1120000	4.310204	0.331822	0.691307
	1140000	4.410767	0.262741	0.836564
	1160000	4.751768	0.290291	0.905895

method	size	sorted	reverse_sorted	random
	1180000	4.814604	0.312328	0.699333
	1200000	4.775984	0.334998	0.735719
	1220000	4.955630	0.295523	0.724457
	1240000	5.124683	0.310603	0.733953
	1260000	5.076342	0.363330	0.796121
	1280000	5.004168	0.345360	0.774391
	1300000	5.403249	0.338479	0.744581
	1320000	5.876622	0.347345	0.738230
	1340000	5.568923	0.343363	0.772891
	1360000	5.374583	0.370427	0.885639
	1380000	5.426433	0.407996	0.877391
	1400000	6.905466	0.336499	0.830980
	1420000	5.561618	0.410417	0.952359
	1440000	6.001009	0.427199	0.804874
	1460000	5.891031	0.388086	0.808817
	1480000	5.645247	0.411075	0.904818
	1500000	5.961272	0.425870	0.949671
	1520000	6.221603	0.403333	0.981189
	1540000	6.171422	0.394363	0.899338
	1560000	7.017478	0.375619	0.878226
	1580000	7.005329	0.408321	0.914875
	1600000	6.549177	0.424334	1.039496
	1620000	6.831188	0.413750	0.933398
	1640000	6.681372	0.489351	0.992671
	1660000	6.423698	0.481749	0.982342
	1680000	7.042122	0.400183	0.866477
	1700000	7.398298	0.475114	0.973657
	1720000	6.752365	0.476929	0.912407
	1740000	7.299394	0.422769	1.033417

method	size	sorted	reverse_sorted	random
	1760000	6.898108	0.494410	1.096418
	1780000	7.225687	0.483914	0.990613
	1800000	7.289383	0.466729	1.064943
	1820000	7.144594	0.854543	1.070039
	1840000	7.421105	0.506287	0.964232
	1860000	7.566232	0.517113	0.921485
	1880000	8.712760	0.543301	1.042794
	1900000	7.666214	0.480231	1.149077
	1920000	7.830655	0.546261	1.069503
	1940000	8.059288	0.496657	1.210533
	1960000	8.238897	0.468571	1.152063
	1980000	7.773765	0.586280	1.011570
	2000000	8.132941	0.546665	1.233093
bubble_method_1	**20000**	0.005023	0.004859	0.004197
	40000	0.008887	0.010679	0.009667
	60000	0.013836	0.013825	0.014698
	80000	0.019640	0.024455	0.020900
	100000	0.023813	0.024436	0.027680
	120000	0.033420	0.034079	0.031080
	140000	0.034359	0.038162	0.041777
	160000	0.058885	0.036136	0.038898
	180000	0.072296	0.054039	0.051854
	200000	0.078309	0.044725	0.048963
	220000	0.053450	0.054134	0.058614
	240000	0.078048	0.081032	0.057712
	260000	0.072674	0.072093	0.071777
	280000	0.070606	0.092385	0.067334
	300000	0.072268	0.074883	0.078256
	320000	0.097083	0.089431	0.085718

method	size	sorted	reverse_sorted	random
	340000	0.100547	0.086917	0.114673
	360000	0.085080	0.122713	0.107210
	380000	0.132487	0.111777	0.124566
	400000	0.096140	0.144179	0.115803
	420000	0.144798	0.142144	0.100478
	440000	0.131616	0.145018	0.111481
	460000	0.125025	0.121674	0.151609
	480000	0.121151	0.116011	0.137434
	500000	0.181224	0.140303	0.123514
	520000	0.160445	0.138471	0.128495
	540000	0.166921	0.150031	0.167870
	560000	0.161495	0.171596	0.143286
	580000	0.186074	0.178008	0.199173
	600000	0.174289	0.169896	0.191423
	620000	0.214725	0.165080	0.192305
	640000	0.216438	0.213111	0.184508
	660000	0.225850	0.191231	0.224885
	680000	0.183617	0.218250	0.207065
	700000	0.192926	0.228132	0.202267
	720000	0.227466	0.227732	0.205877
	740000	0.201667	0.226505	0.304044
	760000	0.214968	0.240653	0.305893
	780000	0.216685	0.220006	0.259549
	800000	0.221833	0.216261	0.233582
	820000	0.243390	0.214836	0.268205
	840000	0.200372	0.257612	0.202160
	860000	0.270050	0.251213	0.270102
	880000	0.272001	0.247154	0.285247
	900000	0.313998	0.266065	0.240397

method	size	sorted	reverse_sorted	random
	920000	0.269598	0.258033	0.257297
	940000	0.254061	0.280721	0.240157
	960000	0.265695	0.297710	0.287442
	980000	0.279244	0.292683	0.393532
	1000000	0.285494	0.307465	0.322595
	1020000	0.279959	0.298611	0.348894
	1040000	0.373996	0.337573	0.328557
	1060000	0.289338	0.313336	0.307620
	1080000	0.296789	0.403702	0.344065
	1100000	0.326142	0.333252	0.311951
	1120000	0.299242	0.336494	0.290768
	1140000	0.288422	0.281686	0.415319
	1160000	0.324054	0.280557	0.429880
	1180000	0.379974	0.337010	0.378146
	1200000	0.377529	0.326958	0.401051
	1220000	0.377850	0.387959	0.362065
	1240000	0.375196	0.378313	0.388468
	1260000	0.371148	0.408764	0.379743
	1280000	0.342543	0.363910	0.331732
	1300000	0.353374	0.378621	0.371255
	1320000	0.380552	0.369209	0.435907
	1340000	0.409646	0.423303	0.419832
	1360000	0.349432	0.411424	0.415711
	1380000	0.405745	0.415099	0.336777
	1400000	0.524677	0.386900	0.389423
	1420000	0.372502	0.458963	0.375917
	1440000	0.432661	0.464664	0.402222
	1460000	0.488271	0.460510	0.475842
	1480000	0.378238	0.435879	0.472991

method	size	sorted	reverse_sorted	random
	1500000	0.474348	0.446529	0.426348
	1520000	0.459323	0.399716	0.477123
	1540000	0.385624	0.425260	0.371699
	1560000	0.449200	0.499797	0.428248
	1580000	0.459330	0.411906	0.593506
	1600000	0.507175	0.436328	0.509935
	1620000	0.488153	0.516688	0.472934
	1640000	0.498742	0.533366	0.467452
	1660000	0.528962	0.421001	0.500684
	1680000	0.504746	0.517499	0.462696
	1700000	0.515021	0.461853	0.461602
	1720000	0.432350	0.494444	0.456574
	1740000	0.502415	0.455125	0.497140
	1760000	0.521441	0.481579	0.509614
	1780000	0.544783	0.504570	0.527664
	1800000	0.507760	0.492833	0.499274
	1820000	0.491785	0.930396	0.523247
	1840000	0.504271	0.544530	0.551461
	1860000	0.528733	0.514003	0.462320
	1880000	0.611946	0.522261	0.524810
	1900000	0.612009	0.511580	0.579215
	1920000	0.602269	0.530070	0.533503
	1940000	0.554700	0.558617	0.556309
	1960000	0.590061	0.534295	0.570640
	1980000	0.544742	0.585278	0.598396
	2000000	0.618615	0.538771	0.528556
bubble_method_2	**20000**	0.012113	0.011531	0.011777
	40000	0.024149	0.024133	0.028981
	60000	0.036192	0.035983	0.048548

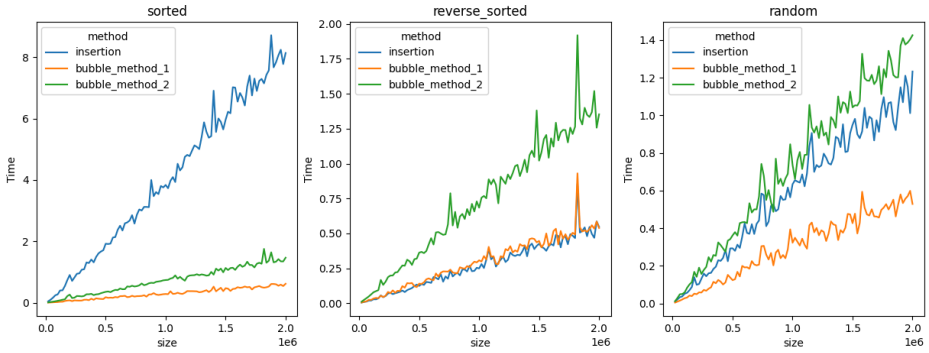
method	size	sorted	reverse_sorted	random
	80000	0.048275	0.048743	0.049726
	100000	0.064407	0.063788	0.058502
	120000	0.080247	0.079136	0.082560
	140000	0.097886	0.086632	0.100519
	160000	0.113050	0.094181	0.113026
	180000	0.217317	0.167157	0.167925
	200000	0.265410	0.131729	0.141253
	220000	0.153602	0.154328	0.189991
	240000	0.166526	0.186853	0.156396
	260000	0.220136	0.198410	0.178010
	280000	0.215722	0.198760	0.193888
	300000	0.207128	0.219981	0.245802
	320000	0.222666	0.223222	0.222232
	340000	0.276823	0.248121	0.257096
	360000	0.276219	0.270629	0.253310
	380000	0.290648	0.268768	0.324201
	400000	0.249256	0.312002	0.309873
	420000	0.273589	0.300646	0.292374
	440000	0.280823	0.274573	0.274353
	460000	0.311592	0.315289	0.333792
	480000	0.334263	0.319841	0.341690
	500000	0.356608	0.362441	0.361796
	520000	0.367294	0.367833	0.378718
	540000	0.346426	0.359116	0.360542
	560000	0.389473	0.374412	0.426538
	580000	0.419310	0.422085	0.431499
	600000	0.412842	0.466133	0.432640
	620000	0.464629	0.420736	0.426516
	640000	0.440694	0.507139	0.532715

method	size	sorted	reverse_sorted	random
	660000	0.443175	0.509933	0.483284
	680000	0.531487	0.499589	0.500057
	700000	0.521231	0.489789	0.499830
	720000	0.554019	0.493650	0.568213
	740000	0.529335	0.565123	0.741552
	760000	0.541470	0.787736	0.678816
	780000	0.567288	0.557114	0.550200
	800000	0.627438	0.654219	0.602639
	820000	0.571415	0.540411	0.538619
	840000	0.594390	0.621420	0.487027
	860000	0.634034	0.639494	0.769387
	880000	0.650541	0.603547	0.635437
	900000	0.643003	0.674575	0.662643
	920000	0.681622	0.630842	0.624542
	940000	0.693390	0.711593	0.663400
	960000	0.705837	0.657181	0.688918
	980000	0.730737	0.729265	0.846080
	1000000	0.739566	0.684362	0.736246
	1020000	0.745811	0.756042	0.661394
	1040000	0.788087	0.768383	0.761151
	1060000	0.808881	0.751683	0.804921
	1080000	0.891378	0.887423	0.714819
	1100000	0.783965	0.851749	0.789078
	1120000	0.844432	0.886913	0.791065
	1140000	0.773737	0.847948	1.055380
	1160000	0.718418	0.715783	0.935352
	1180000	0.850480	0.906183	0.908257
	1200000	0.887727	0.881028	0.941117
	1220000	0.895051	0.855343	0.879119

method	size	sorted	reverse_sorted	random
	1240000	0.898943	0.922058	0.969899
	1260000	0.929657	0.889564	0.892608
	1280000	0.896718	0.931414	0.908410
	1300000	0.924279	0.982126	0.844768
	1320000	0.907845	0.990972	0.990598
	1340000	0.883799	0.910559	0.973156
	1360000	0.894283	0.960735	0.933700
	1380000	1.004486	1.027992	1.100496
	1400000	0.916517	0.938308	1.013604
	1420000	1.111411	1.059661	1.070108
	1440000	1.049401	1.092500	1.060801
	1460000	1.112308	1.047225	1.012130
	1480000	0.992931	1.380475	1.126300
	1500000	1.133065	1.021540	1.042372
	1520000	1.065743	1.081779	1.053197
	1540000	1.129666	1.175270	1.051998
	1560000	1.139589	1.204977	1.075418
	1580000	1.070905	1.041918	1.327138
	1600000	1.174506	1.181043	1.198187
	1620000	1.241453	1.121547	1.184999
	1640000	1.171121	1.292619	1.183722
	1660000	1.212478	1.165987	1.215094
	1680000	1.335658	1.223130	1.165872
	1700000	1.230413	1.241327	1.185357
	1720000	1.228424	1.240343	1.260929
	1740000	1.150554	1.151690	1.112657
	1760000	1.269256	1.257288	1.245566
	1780000	1.303002	1.212068	1.202033
	1800000	1.269456	1.265160	1.343047

		sorted	reverse_sorted	random
method	**size**			
	1820000	1.753532	1.919305	1.293799
	1840000	1.304653	1.324255	1.214457
	1860000	1.368669	1.280157	1.201963
	1880000	1.642718	1.399521	1.201936
	1900000	1.316731	1.349633	1.370024
	1920000	1.337499	1.333692	1.410695
	1940000	1.420674	1.368352	1.376338
	1960000	1.348745	1.519797	1.387152
	1980000	1.346378	1.256971	1.403857
	2000000	1.467212	1.352686	1.425900

Cases Graphed with Functions



The table is listed below:

In [106…
```
df.T
```

Out[106…

	method							
	size	**20000**	**40000**	**60000**	**80000**	**100000**	**120000**	**140000**	**1600**
	sorted	0.045537	0.098389	0.163001	0.243311	0.274680	0.401513	0.405875	0.5472
	reverse_sorted	0.004663	0.009318	0.011982	0.019163	0.019555	0.028707	0.027991	0.0354
	random	0.008509	0.021394	0.032811	0.037184	0.051603	0.056209	0.069824	0.0874

Estimation of Asymptotic Time Through Regression

In [107…

```
import numpy as np
from scipy.stats import linregress

sizes = [20000*x for x in range(1, 101)]
for method in ['insertion', 'bubble_method_1', 'bubble_method_2']:
    for case in df.columns:
        m, b, _, _, _ = linregress(np.log(sizes), np.log(df.xs(method)[case]))
        print(f'{method} {case} slope: {m}')
```

```
insertion sorted slope: 1.1059210527665222
insertion reverse_sorted slope: 1.0591775875284313
insertion random slope: 1.0474240078738088
bubble_method_1 sorted slope: 1.0288813976889206
bubble_method_1 reverse_sorted slope: 1.0320753508263623
bubble_method_1 random slope: 1.0479284543668155
bubble_method_2 sorted slope: 1.0144507975853638
bubble_method_2 reverse_sorted slope: 1.0378875239466805
bubble_method_2 random slope: 1.0064983162969732
```

We can see from the above fitted regression models to the logorithms of each variable that almost all slopes m are close to 1. This helps show that the big-Theta runtimes for each case and function is $\Theta(n)$. The one exception is for the `Inverted` method on the sorted case, where the slope is higher, and therefore can be said to be of $\Theta(n \times logn)$.

Summary

In conclusion, we can see the different results produces by our three methods of heapification (`Insertion` , `Bubble Method 1` , `Bubble Method 2`) through the benchmarking and regression analysis performed in this report. The benchmarking demonstrated a strong linear relation for every method and case, except for `Insertion` and its sorted case which needed more investigation. This was later proven by our regression analysis, which showed that the latter combination can be said to run in $\Theta(n \times logn)$, which the graph supports by being the worst combination of all, reaching beyond **8 seconds** of runtime when compared to sub **2 seconds** for the other algorithms on the same sorted list.

What's further interesting is the fact that `Bubble Method 1` only does better than `Bubble Method 2` on the sorted and randomly shuffled list, whereas they do about the same on a reverse sorted list. This comes as a surprise, as generally the reverse sorted list should be `Bubble Method 2` 's worst case. This leaves room for discussion (which will happen in the following questions).

In [108…

```
print(df.loc[('insertion',2000000)])
print(df.loc[('bubble_method_1',2000000)])
print(df.loc[('bubble_method_2',2000000)])
```

```
sorted            8.132941
reverse_sorted    0.546665
random            1.233093
Name: (insertion, 2000000), dtype: float32
sorted            0.618615
reverse_sorted    0.538771
random            0.528556
Name: (bubble_method_1, 2000000), dtype: float32
sorted            1.467212
reverse_sorted    1.352686
random            1.425900
Name: (bubble_method_2, 2000000), dtype: float32
```

1. **Does the empirically-determined runtime match the theoretic run time? If not, what might have caused the discrepancy?**

For the most part, yes. For both `Bubble Method 1` and `Bubble Method 2`, we got linear slopes, which is what we expected ($\Theta(n)$) for each case no matter what. The interesting part comes with `Insertion`. While the best case is confirmed by the empirical data of regressive analysis, having a linear slope, it is only slightly larger than 1 for the worst case, a sorted list; however, it is still a significant part larger than linear, making it fall into $\Theta(n \times logn)$. This can be confirmed by the graph, which for that specific combination (`Insertion` and a sorted list) has the biggest runtimes and a not-so-linear relationship.

2. **Was there a noticeable difference between sorted, reverse-sorted, and random data? If so, describe the difference and explain why you think it exists**

`Insertion`: This approach did by far the worst in a sorted list. This confirms with why, since it had to bubble up each element in order all the way to the top, as each consecutive element being inserted will be the largest in the heap up to that point. The alternate stands true as well, as the best case (reverse sorted) did noticiably better, even when compared against randomly shuffled.

`Bubble Method 1`: An interesting observation about this approach is that there was little noticiable difference between any of the cases, proving this to be an effective heapification method which keeps runtimes low (hence, it being taught to us). It is important to note, though, that its worst case (sorted list) is running a good percentage longer than the other two cases, as seen by the statement printed above of the last element for each approach.

`Bubble Method 2`: Once again, this approach almost had no noticiable differences between its 3 cases. A worthy note is that this approach was significantly slower than `Bubble Method 1`, and hence should usually not be used. Furthermore, as the note for `Bubble Method 1` stated, this approach's best case (reverse sorted) ran slightly faster than both the sorted and randomly sorted lists, which proves our theory for that being the best case.

3. **What runtime effect do you get by using heapify vs. adding all elements to an empty heap? Explain!**

Adding all elements to an empty heap (`Insertion` approach) did just as well on its best case (reverse sorted) as, say, `Bubble Method 1` , which we determined to be the best heapification approach. However, as the case for `Insertion` gets worse, we can see the time grow at some rate and increase much faster than our heapifications. When it comes to the actual resulting heap, both of the heapification approaches produced the same heaps, while `Insertion` produced some other heap variation *which was still correct*.

4. **Compare bubbling down and stopping short to bubbling all the way down, then back up. When might each approach be better?**

When we do the former (`Bubble Method 1` / the one we did in class), we not only choose the bigger child to swap with, but we do an additional check to make sure that the value is smaller than its children in the first place, which gives us that "stopping short" characteristic. This check is removed on the way down for the latter approach (`Bubble Method 2`), but instead we don't have to check anything other than the inverse of that on the way back up, where we stop at a point when the value's parent is greater than itself. `Bubble Method 1` is good overall since it reduces unnecessary swaps. A good use case for `Bubble Method 2` may be when you have a sorted list, and you know that all values need to be pushed all the way down in order to complete a heapification.