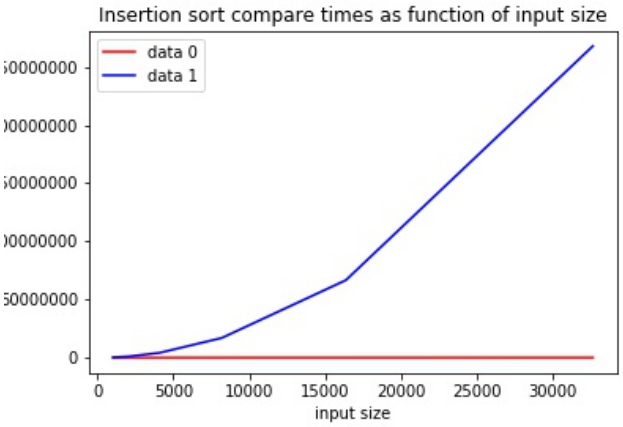
DSA Homework 2 Report

Yiming Bi – yb127 – 146004795

Question 1

Python code is a separate file.
For insertion sort all the way down:



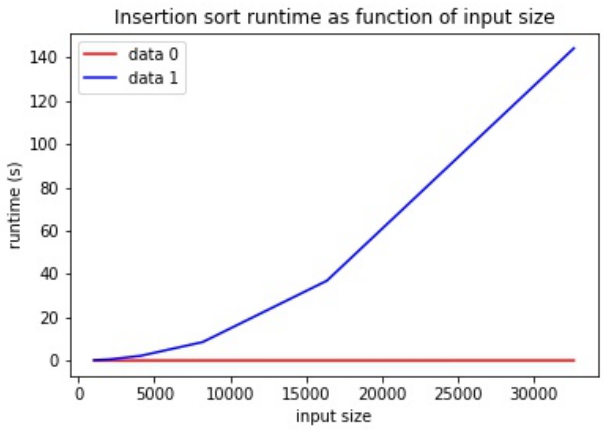


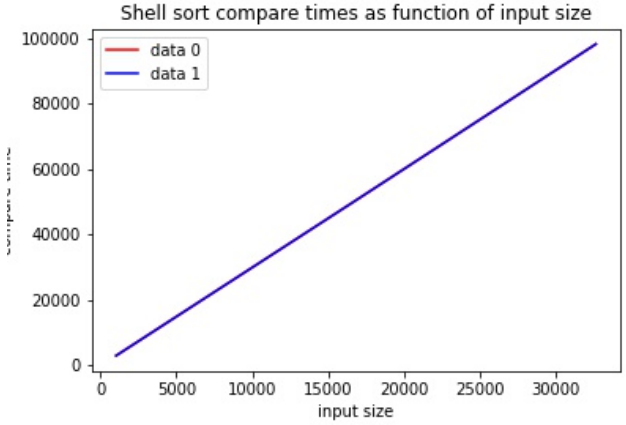
Table of compare times:

Input size	1024	2048	4096	8192	16384	32678
Sorted data	1023	2047	4095	8191	16383	32767
Random data	265553	1029278	4187890	16936946	66657561	267966668

Table of runtime:

Input size	1024	2048	4096	8192	16384	32678
Sorted data	0.000997	0.000998	0.001999	0.002961	0.010008	0.015957
Random data	0.135639	0.528570	2.170209	8.535048	36.884765	144.302265

For shell sort:



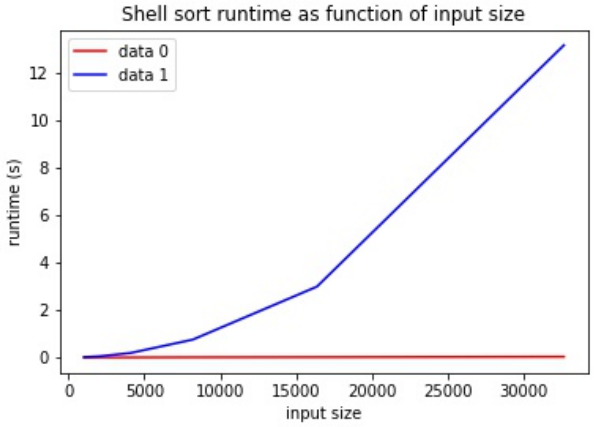


Table of compare times:

Input size	1024	2048	4096	8192	16384	32678
Sorted data	3061	6133	12277	24565	49141	98293
Random data	3061	6133	12277	24565	49141	98293

Table of runtime:

Input size	1024	2048	4096	8192	16384	32678
Sorted data	0.000936	0.003009	0.004031	0.009973	0.015951	0.032913
Random data	0.015620	0.052137	0.192482	0.756979	2.996473	13.197628

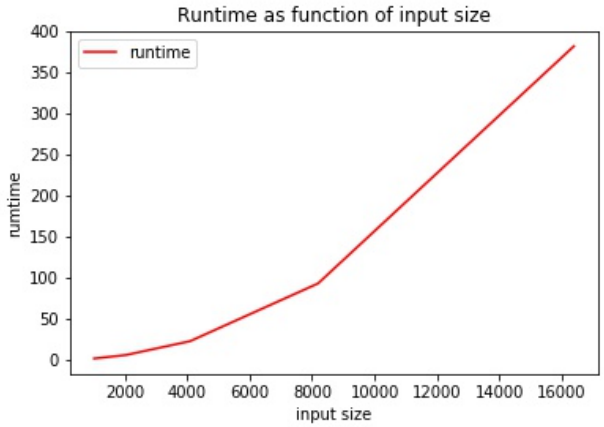
Discussion:

Talking about sorted data and random data, since sorted data is the best case for insertion sort, both compare times and runtime have big difference between two data types. But for shell data, compare times does change while runtime for random data is larger than sorted data,

Between to types of sorting, shell sort has better performance than insertion sort. The reason is that in shell sort, at beginning we use big steps to smaller steps. Bid steps help one element move faster to its correct position in array while insertion sort swap one position each time.

Question 2

Python code is a separate file.



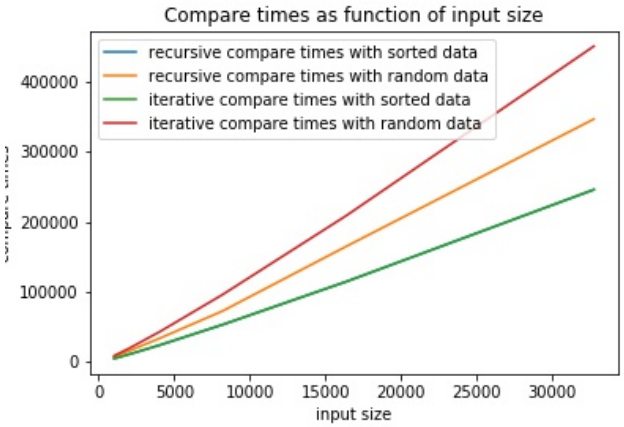
Input size	1024	2048	4096	8192	16384
Runtime	1.434278	5.721009	22.539632	92.852165	381.913227

Runtime for 32768 input size is too long.

Since function uses double for loops, the growth should be in order n^2.

Question 3

Python code is a separate file.



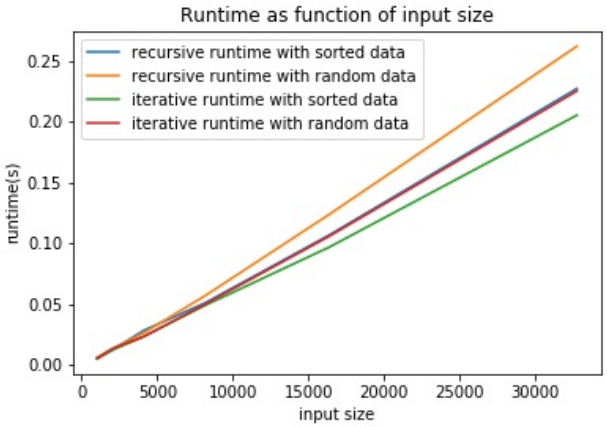


Table of compare times

Input size		1024	2048	4096	8192	16384	32768
Recursive sort	Sorted data	5120	11264	24576	53248	114688	245760
	Random data	7158	16173	34416	72721	164908	346348
Iterative sort	Sorted data	5120	11264	24576	53248	114688	245760
	Random data	8954	19934	43944	96074	208695	450132

Table of runtime

Input size		1024	2048	4096	8192	16384	32768
Recursive sort	Sorted data	0.004983	0.011999	0.027956	0.050864	0.106683	0.227391
	Random data	0.005956	0.012934	0.025931	0.056847	0.123701	0.262303
Iterative sort	Sorted data	0.005018	0.011967	0.022947	0.048869	0.096741	0.205451
	Random data	0.004956	0.013207	0.022938	0.049866	0.105717	0.225398

Discussion:

We can tell that sorted data costs fewest compare times for both sorting.

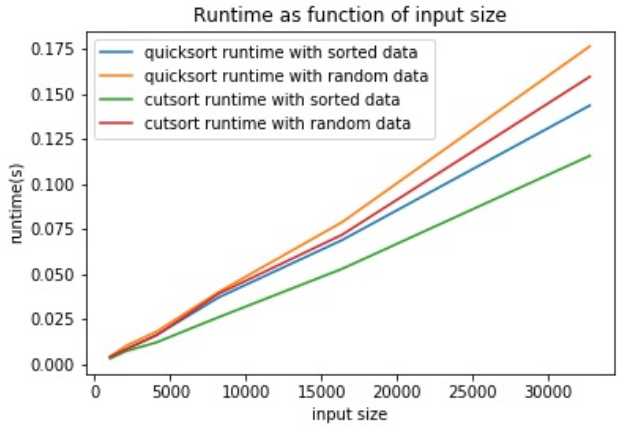
Iterative merge sort costs more compare times but less runtime in general.

Question 4

Since we already know that the data set of 8192 length has certain number of each elements, that means the sort result of any input will be same. In this case, my algorithm will simply give pre-set data with 1024 repeats of 1, 2048 repeats of 11, 4096 repeats of 111 and 1024 repeats of 1111 as the result. Time complexity is in order 1.

Question 5

Python code is a separate file.



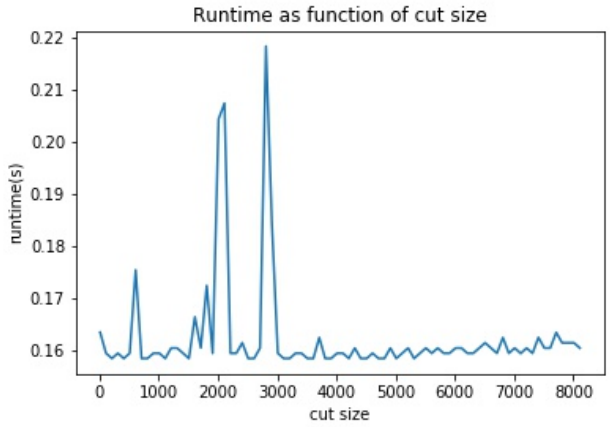


Table of runtime:

Input size		1024	2048	4096	8162	16384	32768
Quicksort	Sorted data	0.004013	0.007984	0.01598	0.036871	0.068846	0.143618
	Random data	0.003989	0.009946	0.017958	0.039894	0.078792	0.176523
Cutsort	Sorted data	0.00299	0.00698	0.011968	0.025898	0.052859	0.115659
	Random data	0.00399	0.00801	0.01596	0.038926	0.071807	0.159605

Discussion:

Quicksort with cut number into insertion sort performs better than quicksort itself in both sorted data and random data conditions. When I change the cut number from 10 to 1/4 of input length with step 100, I can not tell a clear pattern between cut number and runtime. There are some point that cutsort costs more time but under more conditions, cutsort run faster.

Question 6

(1) Knuth shuffle: the column starts with “blue, gray” (column 3) since the elements after “silk” are in the order of the original column while the rest elements are in random order.
(2) Selection sort: the column starts with “bark” (column 7) is selection sort since selection sort finds either the minimum or maximum value first. This column has been sorted till “mint”.
(3) Insertion sort: the column starts with “blue, coal, corn” (column 5) since this column is sorted before “bark” and in the original order after it.
(4) Merge sort (top-down): the column starts with “blue, coal, gray” (column 4) since this is the only column that each continuous three elements are in sorted order. This column matches 24-12-6-3-2-1 merge order.
(5) Merge sort (bottom-up): the column starts with “coal” (column 1) since in this column, each four elements are sorted, which matches 1-2-4-8-16 merge order.
(6) Quicksort (standard, no shuffle): the column starts with “mist” (column 8) since “navy” is in its position and all element before it is smaller while all elements after it is bigger.
(7) Quicksort (3-way, no shuffle): the column starts with “corn” (column 2) since the first, last and medium elements from the original column are sorted in place and rest elements are in first and second halves as the quicksort pattern.
(8) Heapsort: the column starts with “wine” (column 6) since the largest is sorted first.