Sean Fitzgerald
Data Structures and Algorithms
4/4/16

Project 2

Heapsort Results

	10 Elements	100	1,000	10,000
2 Branches	0.0000241	0.0001717	0.0006082	0.0050180
3	0.0000255	0.0001456	0.0006110	0.0042243
4	0.0000262	0.0001387	0.0012310	0.0041069
5	0.0000241	0.0001396	0.0005688	0.0039667
6	0.0000254	0.0001349	0.0005589	0.0047778
7	0.0000238	0.0001295	0.0006268	0.0044279
8	0.0000240	0.0001350	0.0006250	0.0049649
9	0.0000263	0.0001436	0.0005986	0.0055106
10	0.0000261	0.0001372	0.0006298	0.0047683
15	0.0000272	0.0001736	0.0007306	0.0061698
20	0.0000253	0.0001610	0.0007150	0.0067463
30	0.0000294	0.0001949	0.0007508	0.0079891
50	0.0000321	0.0002324	0.0009508	0.0091372
75	0.0000374	0.0002642	0.0011075	0.0112080
100	0.0000400	0.0002874	0.0012720	0.0120146

NOTE: Each value in the table is the average of 10 results and is in units of seconds.

A lot of conclusions can be drawn from the data in this table. First, we can identify the fastest branching factor.

- For 10 elements in an array, branching factors of two and seven are the fastest out of the branching factors listed, with 0.0000241 seconds the average time to sort the array.
- For 100 elements, seven branches is the fastest with 0.0001295 seconds.
- For 1,000 elements, six branches is the fastest with 0.0005589 seconds.
- For 10,000 elements, five branches is the fastest with 0.0039667 seconds.

From these observations, it is clear that the number of elements in the array to be sorted affects which branching factor will result in the fastest heapsort runtime. However, from the above bullet points, we cannot see a clear pattern of change in fastest branching factor between the number of elements. The reason for this is because the fastest runtime is always found amongst the cluster of numbers of branches below ten. These values for numbers of branches are too close together to find consistent runtime results. The reason that values for the number of branches that are close together are not good for finding consistent runtime patterns is because that *any given test of heapsort will not result in the same runtime every time*. I found this by running heapsort ten times for every pair of *number of branches* and *size.* Each runtime was different (this is due to the JRE just not being consistent). Therefore, values for number of branches that are close together may yield inconclusive results. To find a pattern, we must look at the entire table.

For each number of elements given for the array, the runtime generally increases as the number of branches increases. Because this applies to every value of *size* that I passed to heapsort, it seems that, in general, the more branches there are the more time it takes heapsort to

run. Given this fact and the fact that the fastest number of branches is, let's say, an arbitrary value below ten, we can say that the fastest branching factor does not significantly change depending on the number of elements in the array.

The reason that adding more branches to the heap makes heapsort run slower is because it takes longer for heapify to find the largest child. I used a for loop to find the largest child for the given index, so the more children there are the longer it takes the for loop to run. The for loop can't cut itself short, either, because the largest child could be the last one. Furthermore, heapify is recursive, and therefore the effect of iterating one more time over the children compounds itself, because each call of heapify will have to run this extra iteration. Therefore, the more branches in the heap, the slower it will be. From the table it is clear that this effect cannot be observed unless the values of the number of branches are very different, i.e. all the values under ten produced no real pattern. This shows that the effect of increasing the number of branches by one does not necessarily overcome runtime inconsistencies.

It is important to note that some values in the table seem out of place, such as the runtime for four branches with 1,000 elements. These values are the average of 10 results, each of which can vary by a lot. This particular value was averaged from ten values, two of which were much larger than the other values. I do not know exactly why the runtime is not the same every time – other than to say computers are not perfect machines (they were made by humans after all) and cannot expect to take the same amount of time for every process.