SE 2XB3 Group 4 Report 5

Huang, Kehao
400235182
huangk53@mcmaster.ca
L01

Jiao, Anhao
400251837
jiaoa3@mcmaster.ca
L01

Ye, Xunzhou
400268576
yex33@mcmaster.ca
L01

26 February 2021

1 Building Heaps

1.1 Estimations

`build_heap_1` A loose upper bound is easy to establish since `sink` is $\mathcal{O}(\lg n)$ and less than n nodes are non-leaf nodes. Therefore, this algorithm takes at most $\mathcal{O}(n \lg n)$. However, to develop a tight upper bound, notice that different `sink` calls operate on "mini-heaps" of different n. For example, for the first non-leaf node, the height, or $\lg n$, of the heap containing this node and it children is only 1. The complexity of all `sink` operations on a heap with n non-leaf nodes can be expressed as the sum of a series:

$$\sum_{h=0}^{\lg n} 2^h(\lg n - h) = 2n - \lg n - 2$$

Each level of height h has at most 2^h nodes. Sinking each node on the hth level takes at most $\lg n - h$ swaps. Therefore, the tight bound of `build_heap_1` is concluded to be $\mathcal{O}(n)$.

`build_heap_2` Assume appending a node to the bottom of the heap takes the amortized time $\mathcal{O}(1)$. The complexity of the `insert` operation is the complexity of `bubble_up/swim`, $\mathcal{O}(\lg n)$. Since n nodes would be inserted to build the heap, a loose upper bound of this heap building algorithm is $\mathcal{O}(n \lg n)$.

`build_heap_3` One round of calling `sink/heapify` on every node has complexity of $\mathcal{O}(\lg n)$ `sink` operations times n nodes, $\mathcal{O}(n \lg n)$. Each round of `sink` operations is able to move a node up only one level in the heap. In the worst case, for the actual root (maximum/minimum) of the heap to travel from the bottom level to the top level, $\lg n$ (the height of the heap) rounds of `sink` operations are required. Though there is also a helper function `is_heap` in each round of the n `sink` operations, contributing a $\mathcal{O}(n)$ complexity to `build_heap_3`, it is on a smaller scale compared to the complexity of the `sink` operations. Thus, the expected complexity of this heap building algorithm is $\mathcal{O}(n(\lg n)^2)$.

1.2 Experiments

The results of empirical timing experiments was unexpected. `build_heap_2` was surprisingly slow. The run time of `build_heap_2` was around the order of 10^{-4} while the others were of the order 10^{-7}. The different is so large that the scatter plots of the other algorithms were suppressed down to a straight line when plotted along with `build_heap_2`, as shown in Figure 1.

The plotted data for both `build_heap_1` (in Figure 2) and `build_heap_3` (in Figure 3) is rather unstable. Even though we tried to mitigate the effects of undesired factors like system resources reallocation by running multiple tests on one input size and take the fastest run time as the result for the input size, the regressions modeled with our estimation did not fit well with the data.

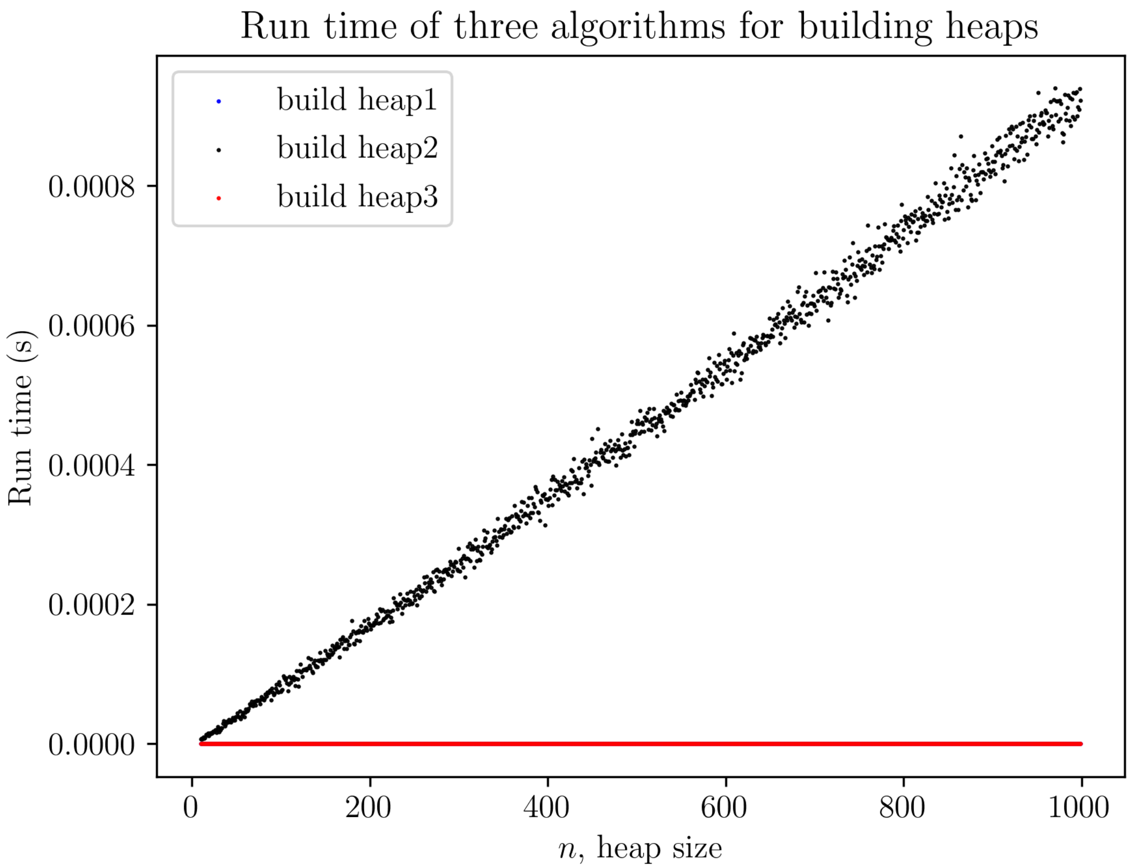

Figure 1: Heap building algorithms comparison

As for `build_heap_2` (in Figure 4), the data suggested a linear growth. This is also different from our prediction. One of the proposed reason is that inserting a list of elements into an empty heap is essentially copying an array, along with the `bubble_up` operation. In empirical timing, the copying or appending action is a lot slower than the element swapping involved in the `bubble_up` operation. Therefore, the $\mathcal{O}(n)$ copying complexity dominates the empirical run time because its run time is at a lower magnitude.

1.3 `build_heap_3` Improvement

The depth of the heap causes the poor performance. If the larger elements are at the bottom of the heap, we would need to heapify the nodes multiple times to get the heap structure. The number of heapifying depends on the depth of the heap.

Improvement: Decreasing the depth of the heap can reduce the number of heapifies. The more children each node can obtain, the shallower the heap depth would be. In this way we could minimize the number of times moving the biggest elements from bottom to top. Consequently, the performance of `build_heap_3` will be improved.

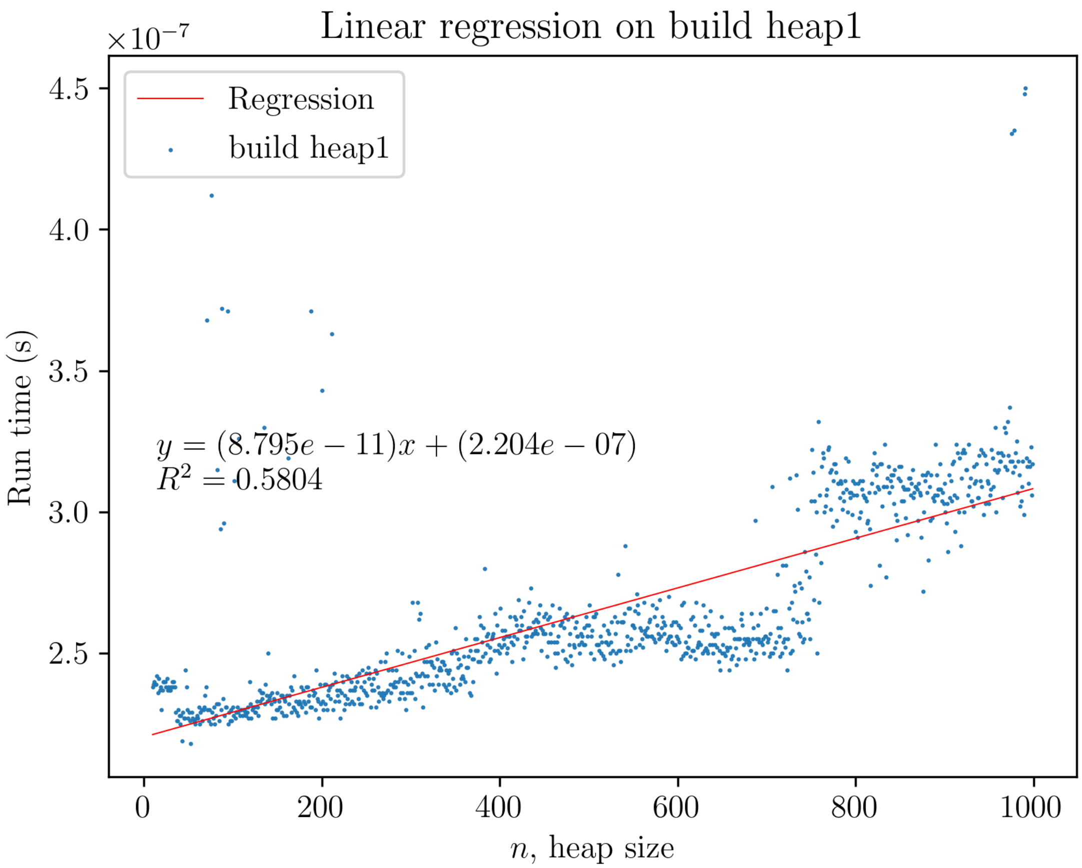

Figure 2: `build_heap_1` run time

2 k-Heap

The asymptotic complexity of `sink` is believed to be $\mathcal{O}(k \log_k n)$. In a k-heap, the nodes are organized in a complete k-ary tree of height $\log_k n$. In the worst case, a node e, needs to "sink" through $\log_k n$ levels. On each level, k comparisons are required to find the maximum element on the level. The maximum would then be swapped with node e. Hence, the complexity of `sink` is k comparisons times $\log_k n$ levels, $\mathcal{O}(k \log_k n)$.

A k-heap is essentially a k-ary tree. The height of a k-ary tree is $\log_k n$, which is smaller than that of a binary tree (heap), $\lg n$. The `swim` operation on a k-ary heap is $\mathcal{O}(\log_k n)$. On the other hand, a larger k results in requiring more comparisons for `sink` on each level of the tree. Therefore, in cases where applications depending solely on the `swim` operation are prioritized over all other operations, k-heaps of a large k have a huge advantage over binary heaps. In other cases where `sink` or both `swim` and `sink` are heavily used, such as Heapsort, k-heaps of any k are not significantly better than binary heaps. In fact, the function family $y = k \log_k x$ minimizes for $k \in \mathbb{N}$ at $k = 3$, which means 3-ary heap is better than binary heap in most aspects, and any k-heap of $k > 3$ trades off its `sink` performance for that of `swim`.

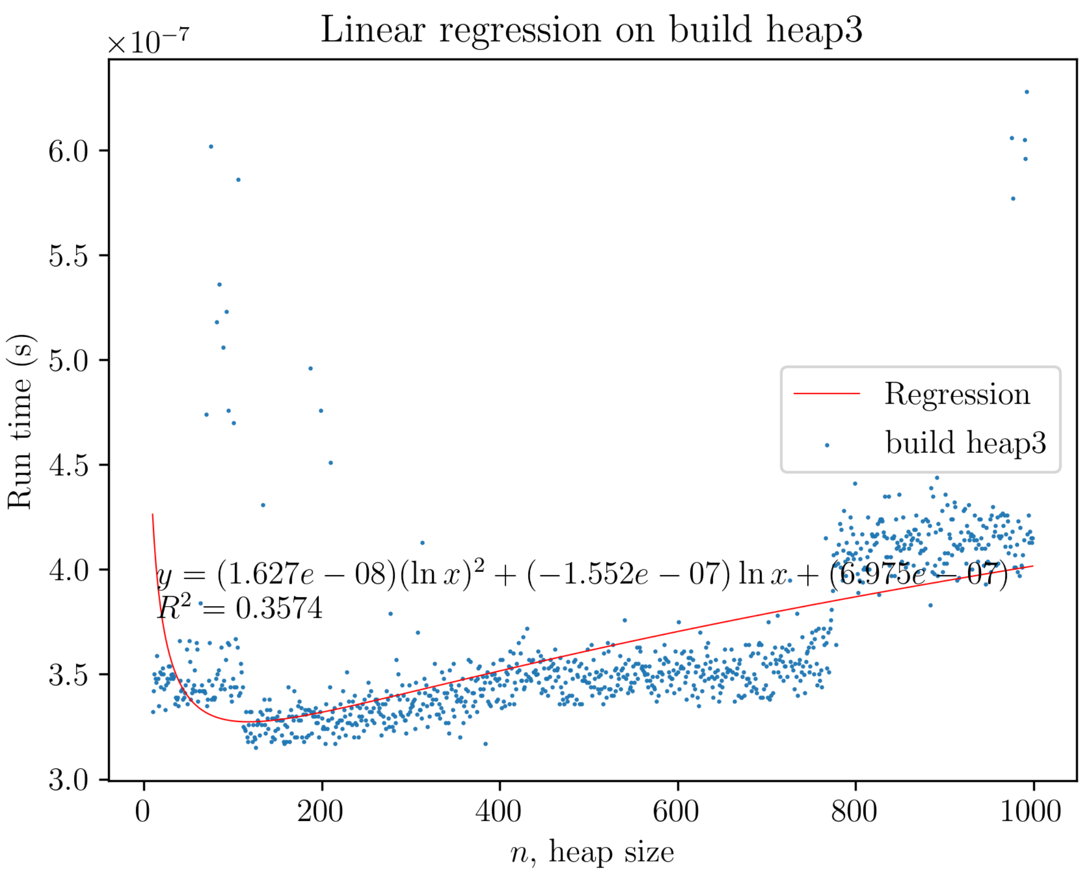

Figure 3: `build_heap_3` run time

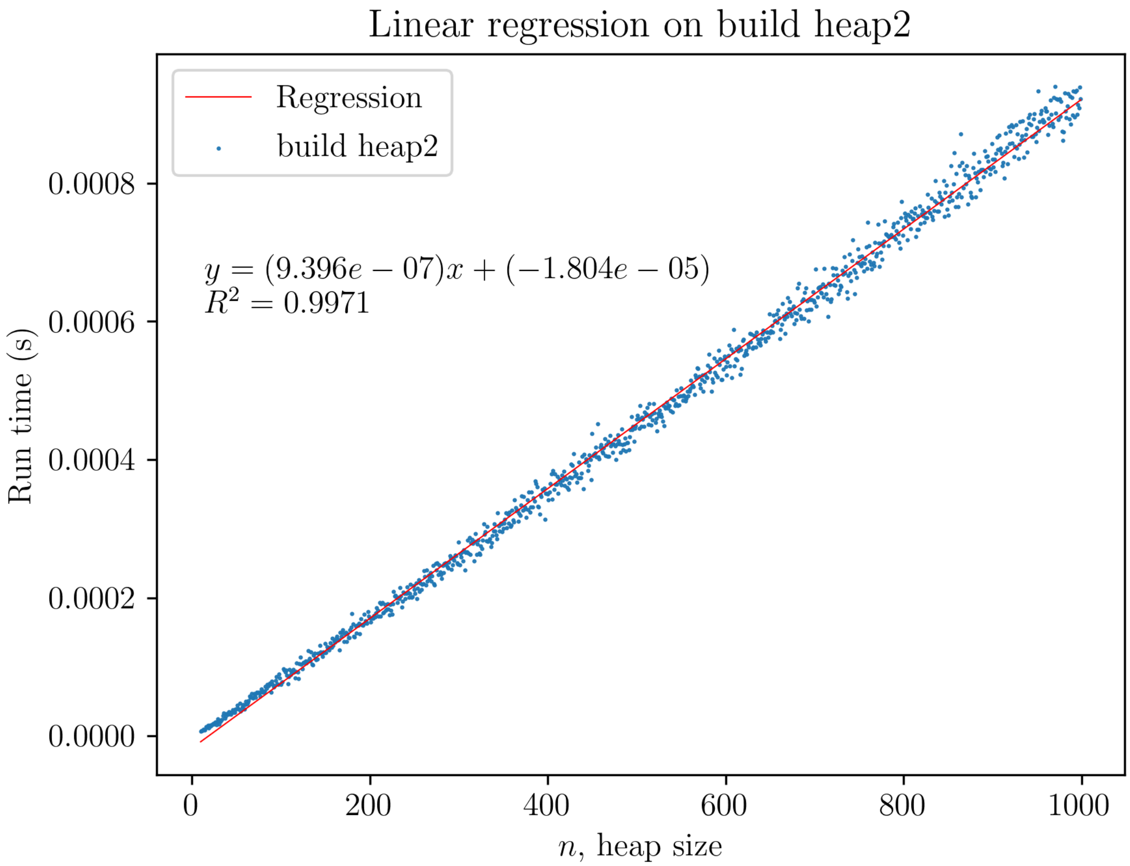

Figure 4: `build_heap_2` run time