CS360 - Assignment 3

6.4-3 What is the running time of heapsort on an array A of length n that is already sorted in increasing order? Decreasing order?

If the array is in *increasing* order, then it is the *worst case* for BUILD_MAX_HEAP (i.e. the largest values are at the bottom of the tree). Hence BUILD_MAX_HEAP will take roughly the upper bound running time of $\approx 2n = O(n)$. However the element extraction phase will still require $O(n \lg n)$ operations so the total running time will be $O(n) + O(n \lg n) = O(n \lg n)$.

If the array is in *decreasing* order, then it is the *best case* for BUILD_MAX_HEAP (i.e. the array is <u>already</u> a max-heap). Therefore it will only take $\lfloor \frac{n}{2} \rfloor$ checks (with no swaps) to verify. Therefore it will take $O(\lfloor \frac{n}{2} \rfloor) = O(n)$. However the element extraction phase ***will still*** require $O(n \lg n)$ operations so that the total running time will still be $O(n) + O(n \lg n) = O(n \lg n)$.

Even though the best case (decreasing) will run slightly faster in practice due to less work to BUILD_MAX_HEAP, the extraction phase asymptotically dominates the algorithm giving roughly the same *asymptotic* running time.

7.2-3 Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.
Initially we note that the pivot element will be the *smallest* element in the array. Hence the first call to PARTITION() will cause lines 4-6 (the **if** block) to not execute since the pivot element is less than all the other element in the array. At the conclusion of PARTITION() the pivot will be swapped with the first element in the array, i.e. the *smallest* element will be swapped with the *largest* element. For example, after the first call to PARTITION() the following array will be

$$< 8, 7, 6, 5, 4, 3, 2, 1 > \Rightarrow < 1, 7, 6, 5, 4, 3, 2, 8 >$$

The first recursive call to QUICKSORT() will be empty (since the pivot moved to the first element) and the second recursive call to QUICKSORT() will contain the remaining $n - 1$ elements (note they are **NOT** sorted at this point).
The next call to PARTITION() will select the pivot element which is the *largest* one in the array. Hence lines 4-6 (the **if** block) will execute for *every* iteration of the loop since the pivot element is greater than all the other element in the array. However the body of the **if** block will simply exchange each element with itself and then PARTITION() will complete by swapping the pivot element with itself. Hence after the second call to PARTITION() the array above will be

$$< 1, 7, 6, 5, 4, 3, 2, 8 > \Rightarrow < 1, 7, 6, 5, 4, 3, 2, 8 >$$

The first recursive call to QUICKSORT() will now contain elements $A[2..n-1]$ (note they **are** sorted in decreasing order at this point). The second recursive call, however, will be empty. PARTITION() will behave similar to the initial case resulting in the second smallest element being moved to the second location in the array as

$$< 1,7,6,5,4,3,2,8 > \Rightarrow < 1,2,6,5,4,3,7,8 >$$

The next pass will not change the array (since the next to last element is the second largest one). This pattern will continue causing one of the two recursive calls to QUICKSORT() to always be empty giving the recursive equation as

$$T(n) = T(n-1) + O(n)$$

which is identical to the worst case behavior when the initial array is sorted in increasing order giving (even though the operation of the algorithm is slightly different)

$$T(n) = \Theta(n^2)$$

Q3. One way we can circumvent the $\Omega(n \lg n)$ bound is if we know something about the distribution of the data prior to sorting. Suppose a set of n data points is provided such that it is clustered into partitions of size k (hence there are n/k partitions) where all the values in a partition are *less than* all the values in the next partition (and *greater than* all the values in the prior partition) but the values *within* a partition are *not* sorted. Thus to sort the entire input it is sufficient to simply sort each partition. Show that a lower bound for sorting this type of "batch"-ordered input is $\Omega(n \lg k)$.

Since each partition has k elements, there are $k!$ permutations of each of the n/k partitions. Hence the total number of permutations of the *entire* input is the *product* of the individual partitions (since they are independent of each other)

$\underbrace{(k!)(k!)...(k!)}_{n/k} = (k!)^{n/k}$

These constitute the 2^h leaves of the decision tree $\Rightarrow 2^h \geq (k!)^{n/k} \Rightarrow h \geq \frac{n}{k}\lg(k!)$.

Then

$$\begin{aligned}
\frac{n}{k}\lg(k!) &= \frac{n}{k}\Theta(k \lg k) \quad \text{(e.q. 3.19)} \\
&\Rightarrow \frac{n}{k}c_1(k \lg k) \leq \frac{n}{k}f(n) \leq \frac{n}{k}c_2(k \lg k). \quad \text{(def. of } \Theta) \\
&\Rightarrow c_1(n \lg k) \leq \frac{n}{k}f(n) \leq c_2(n \lg k) \\
&\Rightarrow \frac{n}{k}f(n) = \Theta(n \lg k) \Rightarrow \Omega(n \lg k)
\end{aligned}$$

Therefore

$$\begin{aligned} h &\geq \frac{n}{k}\lg(k!) \\ &= \Omega(n \lg k) \end{aligned}$$

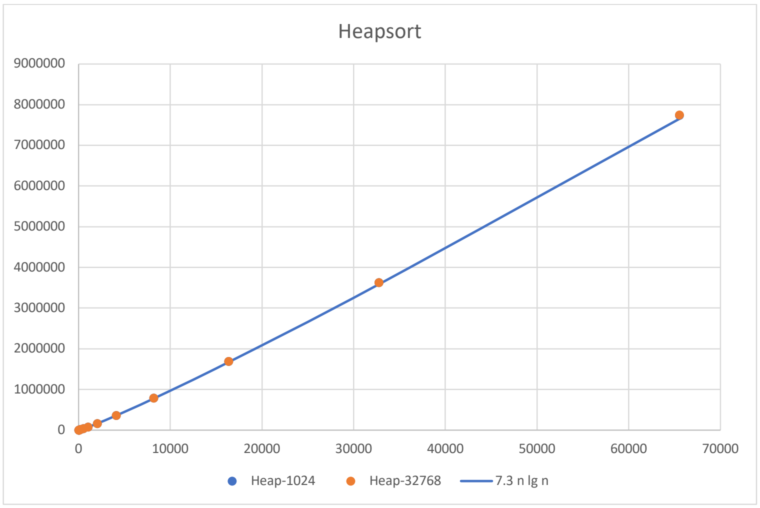
Implementation:

Is there a table showing the empirical data for both element ranges (but it should *not* contain the calculated trend values)?

Is there a graph **for each sort** with:

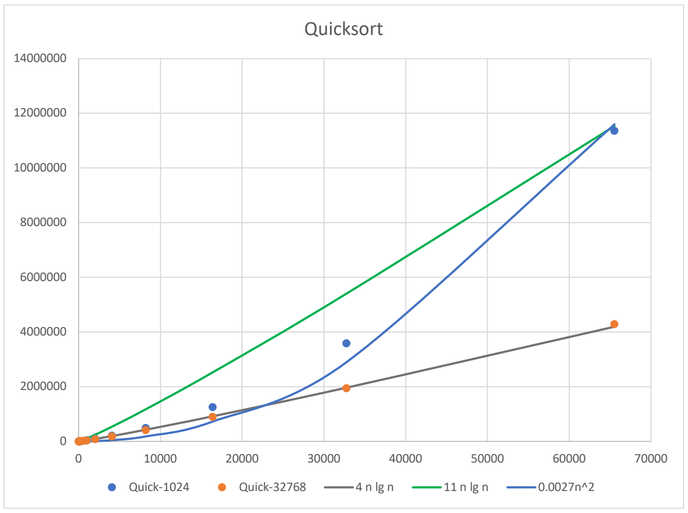
1. The empirical data for both element ranges plotted as **only** *points*
2. The calculated trend for *each* element range plotted as **only** a *curve*
3. A legend listing each data set and showing the trend curve function with the approximated *constant*
4. Properly labelled graph axes

HeapSort



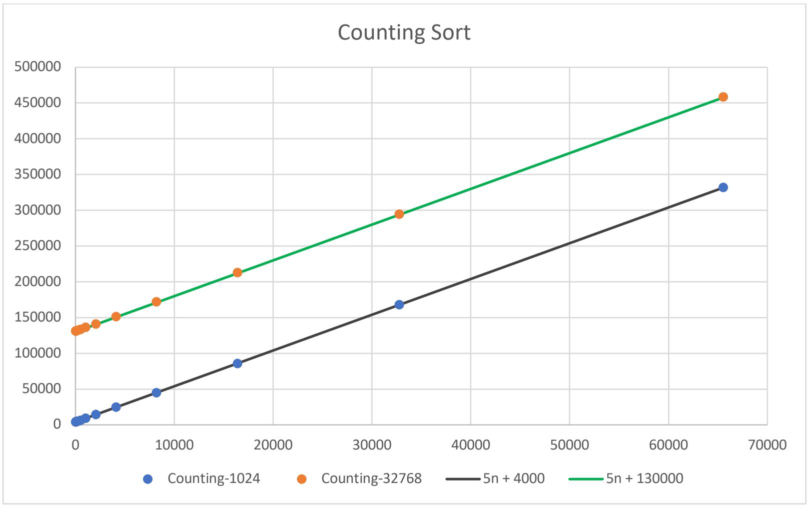
Constants - HeapSort: $1024 \approx 7.3n\lg n$, $32768 \approx 7.3n\lg n$

QuickSort



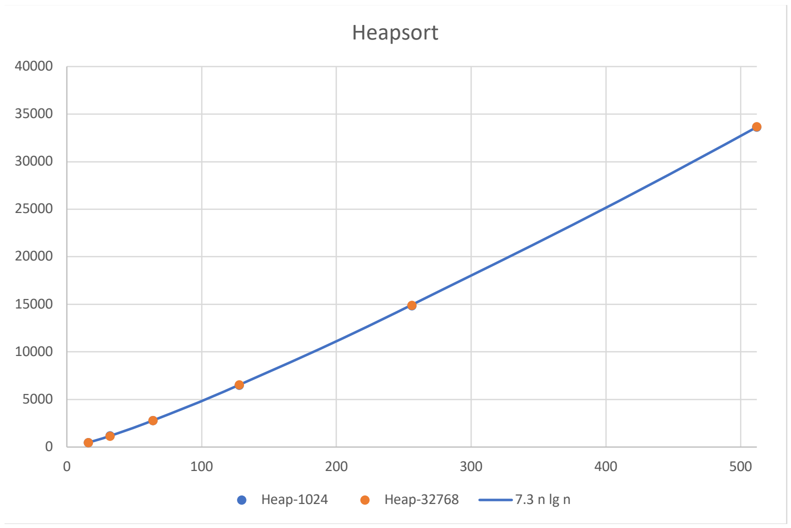
Constants - QuickSort: $1024 \approx 11n \lg n$ **(or** $0.0025n^2$**)**, $32768 \approx 4n \lg n$

Counting Sort

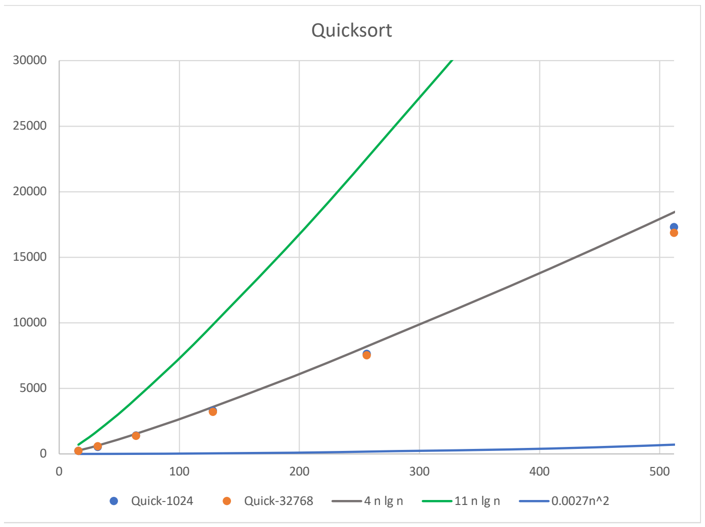


Constants - CountingSort: $1024 \approx 5n + 4000$, $32768 \approx 5n + 130000$

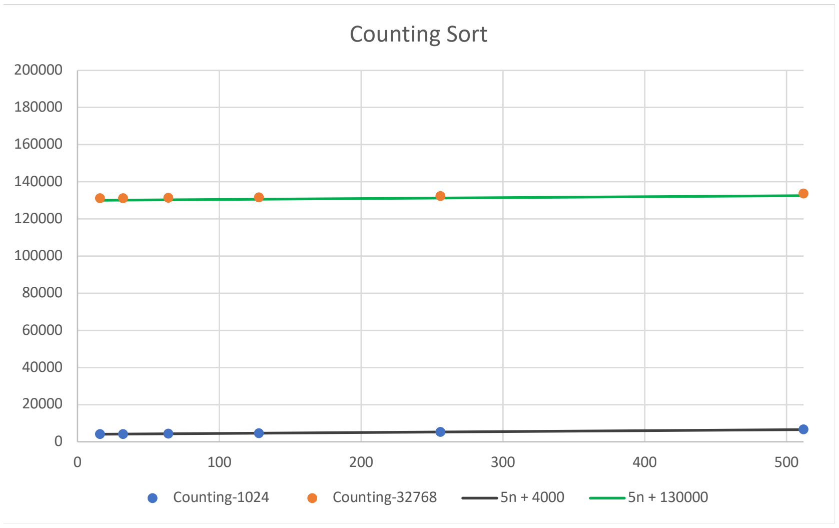
Small Data Sets



The graph above illustrates that the asymptotic curve for heapsort fits the empirical data relatively well even for small data set sizes regardless of element range.

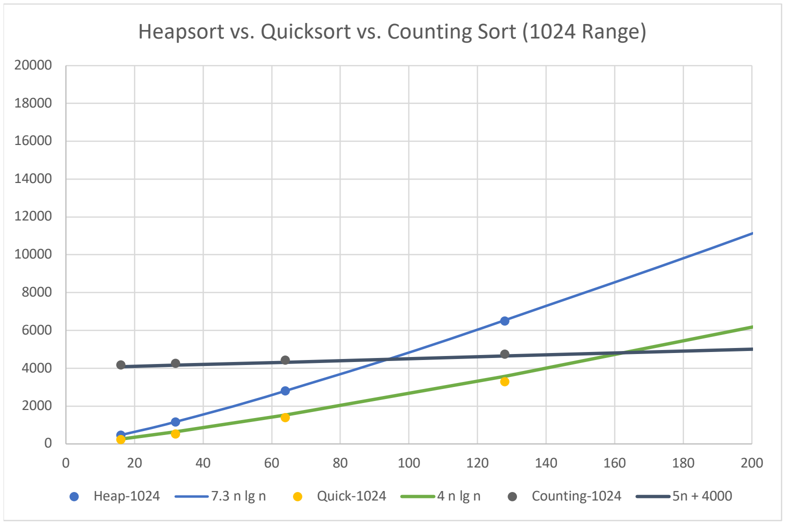


The graph above illustrates that the $4n \lg n$ asymptotic curve for quicksort fits the empirical data relatively well even for small data set sizes. However, the other asymptotic curves that fit the 1024-range data for large n do **not** fit the 1024-range data for small data set sizes. Thus the actual runtime for quicksort is sensitive to small element ranges.

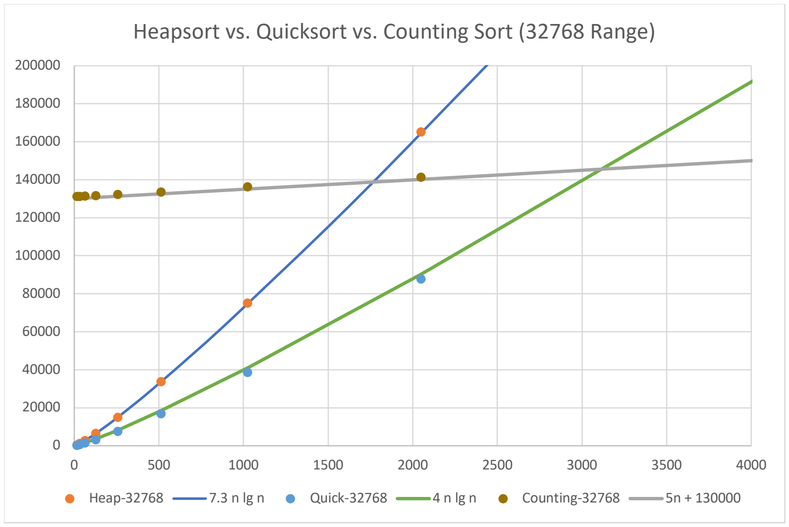


The graph above illustrates that the asymptotic curve for counting sort fits the empirical data relatively well even for small data set sizes regardless of element range.

HeapSort vs. QuickSort vs. Counting Sort



The graph above illustrates that for small element range data, quicksort is the fastest for sets where $n <\approx 160$. HeapSort, while slower than quicksort, is faster than counting sort for sets where $n <\approx 90$. Above $n \approx 160$, counting sort is the fastest with a widening gap as the data set size grows due to the more efficient asymptotic behavior.



The graph above illustrates that for large element range data, quicksort is the fastest for sets where $n <\approx 3000$. HeapSort, while slower than quicksort, is faster than counting sort for sets where $n <\approx 1700$. Above $n \approx 3000$, counting sort is the fastest with a widening gap as the data set size grows due to the more efficient asymptotic behavior.