Robert Barnes

DSA 20407

2013b

Maman 14

Sorting k out of n elements: Comparing Heap Sort with Select Plus Quicksort

Abstract

When trying to find the k smallest elements from a list of n, potentially non-unique elements, we examine the theoretical and empirical performance of two algorithms:

1. Building a min-heap and calling extract k times
2. Calling Randomized Select to partition the array around the kth order statitic, then calling Quicksort on the first k elements.

We find that the heap based solution outperforms the partition based solution in a large number of individual cases, especially as n grows, and k approaches n. This may be due to repeated values which increase the inherent "sortedness" of our inputs, and potential inherent order in psuedo random sequences generated by a Linear Congruential Generator, both of which cause violations of the assumptions upon which the expected average case analysis of quicksort is based.

Results

The tests are run on array's whose elements are generated by the `C` library `rand` function and then reduced to the range 0 to 999 using the modulo function.

The first set of columns is heapsort, the second set select plus quicksort.

n k	3	50	100	3	50	100
100	215	686	1037	153	1304	1951
200	377	1012	1609	210	1319	2417
500	987	1678	2502	676	2333	4640
1000	1923	2757	3654	2557	3842	4007

We find from these results, and numerous additional runs that the constants for

heap are constitently close to $O(1.85 \cdot n + 1.5 \cdot k \lg n)$.

For building the heap, we find that according to the analysis in the book, the theoretical worst case is bounded by:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = n \cdot \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} = 2n$$

We can see that the theoretical best case occurs when the array is already sorted in ascending order, in which case it takes $2 \cdot \frac{n}{2} = n$ time to build the heap.

For extraction, we have in the worst case a time of of $O(1)$ to extract an element and then $O(2 \lg n)$ to restore the heap property (since we must make exactly 2 comparisons at each level of a complete tree), leading to a time of $O(2k \lg n)$ to extract k elements.

In the best case, all elements are identical and it takes $O(2k)$ time to extarct an element and restore the heap property.

We see that the empirical results are highly consistent with the theoretical results, and that on our inputs it performs somewhat closer to the worst case than the best case.

Regarding algorithm 2, the results are more erratic. For $n \leq 1000$, we see that the execution time varies from almost exactly $n + k \lg k$ in the best case, to nearly $4 \cdot (n + k \lg k)$ in the worst case. On average it seems to be close to $2 \cdot (n + k \lg k)$, but the upper bound grows as n increases and k approaches n. These results are consistant with the expected average case runtime of $O(n + k \lg k)$, when running on unique, "random" input, however they exceed the constant of 1.386 calculated by Hoare. Why? The average case analysis depends on two assumptions, namely, that the input is randomized and the values are unique. Clearly not all the values are unique. When using a fixed pivot, quicksort performs poorly on sorted, or nearly sorted data. Adding repeated values increases the "sortedness" of the array and causes the performance of fixed pivot quicksort to deteriorate. In addition, the supposedly "random" data generated by the `C` library `rand` function seems to have a significant amount of order in it.
This library function generates psuedo random numbers using a Linear Congruential Generator, which has the property that sets of k sequential values will all appear in parallel hyperplanes in a k dimensional space, indicating some level of order to the psuedo random values. These hypothosis' are supported by empirical testing. Testing on random arrays of unique values produced only minor improvment for $n \leq 1000$ although the improvement becomes more significant as n grows. However, randomizing the pivot choice in quicksort showed dramatic improvement in both situations, with the average case constants approaching $2n + 1.1k \lg k$ for $n \leq 1000$.

Conclusions

Data generated by an PRNG may have significant inherent sortedness to it, and repeated values increase the level of sortedness. Fixed pivot quicksort is a poor choice for this type of data, since as the ratio of unique values to total values decreases, the inherent sortedness of the data increases and performance deteriorates. This effect can also be seen with randomized quicksort, for larger n where this ratio is less than $\frac{1}{4}$. On the other hand, heapsort provides consistent performance accross all inputs within tight bounds and may be preferable over even randomized quicksort for small values of k, or when dealing with data sets with very low ratios of unique to total values.

Additional Results

S1, S2 = Select + Fixed Pivot Quicksort

S3, S4 = Select + Randomized Pivot Quicksort

Average of 1000 runs:

$n = 1024$

Heapsort, S2, S4: 641/1024 Unique Vaules

S1, S3: 1024/1024 Unique Vaules

k	4	16	64	256	1024
Heap	1978	2193	3046	6373	17314
1.85n + 1.5 k lg n	1954	2134	2854	5734	17254
S1	2066	2250	3194	8306	33771
S2	2053	2270	3247	8701	35128
1.95n + 3.25 k lg k	2023	2205	3245	8653	35277
S3	2074	2288	2840	5286	13348
S4	2086	2211	2841	5268	13356
2n + 1.15 k lg k	2057	2122	2490	4403	13824

$n = 4096$

Heap, S2, S4: 983/4096 Unique Vaules

S1, S3: 4096/4096 Unique Vaules

k	4	16	64	256	1024
Heap	7768	8031	9082	13263	29643
S1	8263	8396	9613	15163	44786
S2	8248	8356	9867	16733	54116
S3	7978	8482	9162	12115	24233
S4	8334	8422	9321	12226	24428