Jack Dubil

2015-10-16

SE/CS 146 Section 01

Project 1 Part 2: Quicksort Report

**Using the Last Element as the Pivot**

Much like how the basic matrix multiplication algorithm was simple to implement, this basic way of implementing quicksort was also very simple. The hardest part was to create the partition algorithm, but even that was not very hard, since it is essentially just a single for loop with an if-statement.

**Using Select and Median of Medians to Use the Median as the Pivot**

This version was MUCH harder to implement, and required some changes to the partition method that I wrote. As the textbook suggested in discussion about this version of the algorithm, I modified partition to take in the element to be used as the pivot as a parameter. However, just knowing the element isn’t enough, the element also needs to be at the end of the array, so I also had to add in a loop that moves the element to the end of the array, since I wasn’t sure how else to handle that situation. As suggested on Piazza, I also used Arrays.sort to handle finding the medians of the groups of five, and when there were less than five elements passed to the select method. It took me quite a while to get the implementation for select correct, mostly due to being slightly off with adding or subtracting a 1 in various places to ensure the correct parts of the array were being worked on.

**Running Times and Comparisons**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array Size | 10,000 | 100,000 | 1,000,000 | 10,000,000 | 100,000,000 |
| Time for QS1 (ms) | 3 | 12 | 133 | 1,557 | 17,988 |
| Time for QS2 (ms) | 17 | 95 | 1,041 | 11,542 | 131,572 |
| Comparisons in QS1 | 172,085 | 2,251,320 | 26,036,187 | 316,773,385 | 3,551,740,466 |
| Comparisons in QS2 | 555,050 | 6,946,623 | 84,358,450 | 989,273,183 | 11,390,616,655 |

**Evaluations**

Obviously, from the running times above, quicksort 2, with the select algorithm, takes quite a lot longer than the simpler method of just choosing the last element in the array as the pivot. This might just be because of my implementation of the median of medians select not being optimal, but I think it is mostly because quicksort 2 does a lot more work than the simpler implementation. Although the second method does guarantee a good partition in using the median, so the case where you have a bad split should be much less likely, it also takes a lot more work and recursive calls to get that median. In addition, in cases such as these, where you have arrays thousands and millions of elements in length, with all of the values randomly generated, it is very unlikely that just choosing the last element of the array will result in a bad split. Thus, I don’t think it is surprising that the simpler implementation of quicksort is actually still faster, since it is just so unlikely to get a bad split, and much faster to choose the last element than go through O(n) selection.

It also makes sense that the median version of quicksort has more comparisons, since the calls to select also rely on using partition, so there are several recursive partition calls just in finding the median, before using that median in the partition call in quicksort itself.

**Conclusions**

Unless you know that you will be working with data that is more likely to result in bad splits, it seems like it is MUCH more efficient to just randomly chose the pivot, rather than go through the whole process of finding the median for every recursive call to quicksort. That implementation could probably be made more efficient than the way I did it, but choosing a random pivot/having random data in large enough amounts seems sufficient to avoiding most issues with having a bad partition.