Richard Idigo

Professor Young

CS 331

May 28th, 2015

**Project 2: Selection Problem with Kth Element**

**Test Strategies**

Since the project was centered around sorting a list with the intent of obtaining the kth element, it was natural to implement the array data structure. We were to implement four different algorithms such that:

***Algorithm 1:*** *Sorts array through merge-sort and returns the kth element*

***Algorithm 2:*** *Uses partition from quick-sort to find the kth element iteratively*

***Algorithm 3:*** *Same as algorithm 2 but recursively*

***Algorithm 4:*** *Same as algorithm 3 but using the median of medians technique to select pivot.*

To start I worked on each algorithm separately as their own programs to focus on each individually. I did this of course for ease of debugging as well as easier tracking of each algorithms. As I verified each was correct I one by one implemented them as methods that I stored in one java program called *SortK.*

To test each of these algorithms against each other, I thought it would be best to dedicate an array to each algorithm for it to perform its operations. However, each array would consist of identical contents so that the integrity of the analysis is upheld. Similar to project one I decided that the System.nano class would be ideal for timing the algorithms in their search for the kth element. So in the beginning each algorithm I take a snapshot of the internal time and subtract it from another snapshot taken at the end to capture the total time in between the two. I decided to also run all the algorithms one after the other to easily compare the results and did so for different values of n. After executing each run 10-20 times I kept the average for each algorithm against each size of n.

**Data Sets**

Project 2 required that while testing each algorithm I used the same data to fairly compare results. Because I wanted to use consistent data, but also have an element of randomness for proper analysis I decided to populate one array randomly with values 1 through 1000. After populating the first array, I created three more arrays that would essentially contain the values of the initial array along the size of the first array. This way I incorporated the necessary randomness and also made sure to keep consistency. I also made sure to vary the sizes of the arrays for instance size 10, 100, 1000 and so on with more powers of 10. I found that at around 100 million for the array length, I would get the following error:

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space at cs331.SortingK.main.

After encountering I took notice that I reach an upper bound and let n be at most 10 million, which is still a pretty impressive size for an array to be sorted. I also found different kth elements, for instance when k=1, k=n/2, k=3n/4. In the chart attached all the data was found when k=3n/4. I thought this would be a great case to study because it wasn’t the first element, last element, or middle. It wasn't a real typical element to consider for arrays and will be dynamic depending on the array. For instance in an array of size 10, 3n/4 produces the 7th element. However in an array of size 10,000 3n/4 produces the 7,500 element. I thought the fact that the element changes dependent on the size of the array would be a great feature to analyze.

**Average Computation of Algorithms (in nanoseconds)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm / n** | **Select kth 1** | **Select kth 2** | **Select kth 3** | **Select kth 4** |
| **n=10** | 77302.8 |  |  | 184987.333 |
| **n=100** |  |  |  | 231511.87 |
| **n=1000** |  |  |  |  |
| **n=10000** |  |  | 2011797.2 | 3092118.211 |
| **n=100000** |  |  | 425413.67 | stack overflow |
| **n=1000000** |  |  | stack overflow | stack overflow |
| **n=10000000** |  |  | stack overflow | stack overflow |

**Explanation of Results**

**Is select kth 2 always faster that kth 3?**

At first when n=10, kth 2 is definitely faster than kth 3 with kth 2 taking about 150000 ns and kth 3 taking almost 2 million ns. However when n =100, kth 2 takes almost 600000 ns while kth 3 takes almost 500000 ns. So with the instance of array size being 100 it is very close. As n gets bigger than 100 kth 3 becomes significantly faster. For instance when n=1000 according to the table we see that the amount of nanoseconds for kth 2 is 4866005.5, compared to kth 3 that took an amount of 292844 nanoseconds.

**When is Select kth 4 faster than kth 1**

In my implementation, it seems that the select 1 is faster until about when n= 10000. I feel this is due to the fact that select 1 still performs well within a certain size n, and within that range the recursion of select 4 is just tedious. However after the range, the recursion of select 4 becomes more efficient therefore making it the better algorithm outside of the range.

**Stack Overflow**

For the algorithms that used recursion, aka select kth 3 and kth 4, I would receive stack overflow errors when the size of the array reached a certain size. I figured this is due to the number of recursive calls that were made. So for certain sizes I couldn’t find the kth element due to this error.

**Strength and Constraints of Your Work**

This project was created and tested with the following hardware and software;

|  |  |
| --- | --- |
| **Hardware**  - Toshiba notebook  - Intel(R) core i3-3120M CPU processor @ 2.50ghz  - ram 4GB (3.88 GB usable)  - 64-bit Operating system type  - 750GB Hard Drive (545 GB usable) | **Software**  -Operating System Windows 8.1  -IDE Eclipse Luna  Version: Luna Service Release 1 (4.4.1)  Build id: 20140925-1800  Programming Language: Java |

**Choice of Hardware-**

I chose to use this particular hardware combination mostly because my Toshiba laptop is portable, meaning I am not restricted to working on the project in one place, such as if I used a desktop. With this decision I was able to work on the project everywhere I brought my laptop including school. The ram space was more than sufficient enough for the work and so was the hard drive space.

**Software Choice-**

The operating system came installed on the Toshiba notebook and I have since then not update the OS. I chose eclipse mostly due to the fact that it is the IDE that I have used most in my programming and the one I am most familiar with. I chose java similarity because it is the programming language I am most familiar with and I program with it regularly.

**Work Conditions-**

While working on the project, I made sure during testing to limit the activities of my laptop as much as I could so it could focus and give its entire attention to the program. However, I did have my Mozilla Firefox browser open more than 95% of the time. In case I ran across an issue I wanted to be able to search the web as necessary for resources.

**Program Correctness**

In terms of optimization I could of course start with optimizing the algorithms I decided to use. For instance for the iteration algorithms I could have structured the code differently. So with that I could have also used better hardware such as an i5 processor instead, or an architecture with more RAM and other hardware components. I also could have tried programming in another language such as Python as it is less “restrictive” than Java.

As far as limits I could have possible tested larger values for n. For instance the largest values I could test up to was array of size 10 million, but that’s because the recursive methods I implemented would cause Stack Overflow errors because of the many recursive calls. Maybe by implementing those algorithms a bit better, I could allow for bigger array sizes and avoid the errors.