Practical Complexity of Standard Sorting Algorithms

Proposed Experimentation Procedure:

This is a practical experiment on the complexity of quick, shell and insert sort. I intend to approach this experiment with the goal of finding notable differences in their move, compare and time factors. With this in mind I decided to State two counters , moveCount and compareCount, as class global variables. These counters were then incremented inside of the sorting method’s code in their respective places. These counters would be stored or printed and then reset for the next function call. This is a similar approach I would take to the time counter. Calling nanoTime before the function call and then subtracting that value from the next nanoTime call after the function finishes. I would then move on to storing the data that would be then printed or plotted. Also calculating averages on different arrays of the same size.

Code for Counting Operations:

Insert and Shell Sort

**private** **static** **int** moveCount = 0;

**private** **static** **int** compareCount = 0;

**while** ((index >= first) && (toSortElement.compareTo(a[index]) < 0))

{

*compareCount*++;

// Shuffle elements over to the right, put firstUnsorted before them

a[index+gap] = a[index];

index = index - gap;

moveMade = **true**;

}

**if** (moveMade) {

//System.out.println("Inserting " + toSortElement + " at pos " + (index+1));

a[index+gap] = toSortElement;

*moveCount*++;

}

Code for Counting Operations:

Quick Sort

**while** (upIndex <= downIndex)

{

// scan right until larger than the pivot

**while** ( (upIndex <= downIndex) && (c.compare(s[upIndex], pivotValue)<=0) ) {

upIndex++;

*compareCount*++;

}

// scan leftward to find an element smaller than the pivot

**while** ( (downIndex >= upIndex) && (c.compare(s[downIndex], pivotValue)>=0)) {

downIndex--;

*compareCount*++;

}

**if** (upIndex < downIndex) { // both elements were found

temp = s[downIndex];

s[downIndex] = s[upIndex]; // swap these elements

s[upIndex] = temp;

*moveCount*++;

}

}

Code for Creating Random Arrays

**private** **static** String[] randomArray(**int** s) {

String[] arr = **new** String[s];

**for**(**int** i = 0; i<s; i++) {

String rand = "" + (**int**)(Math.*random*()\*50);

arr[i] = rand;

}

**return** arr;

}

Code for Running Experiments and Timing

**public** **static** **void** main(String[] args)

{

**double**[] Insertion = **new** **double**[9];

**double**[] Shell = **new** **double**[9];

**double**[] Quick = **new** **double**[9];

String size = JOptionPane.*showInputDialog*("Size of Array to sort:");

**int** s = Integer.*parseInt*(size);

String[] arr = *randomArray*(s);

String[] arr1 = *randomArray*(s); // arrays of same size to test

String[] arr2 = *randomArray*(s);

// Going to sort a copy of the array, not the original.

// Note that it is NOT sufficient to say "String[] copy = arr;" as this

// just copies a new reference to the same array in memory.

System.***out***.println("");

System.***out***.println("Compares || Moves || Time");

// we can also use insertionSort directly

String[] copy1a = arr.clone();

String[] copy1b = arr1.clone(); // clone arrays for use

String[] copy1c = arr2.clone();

**for**(**int** i = 0; i<3; i++) {

*time* = System.*nanoTime*();

*compareCount* = 0;

*moveCount* = 0;

**if**(i == 0) {

*insertionSort*(copy1a);

}

**if**(i == 1) {

*insertionSort*(copy1b); // go through each sort

}

**if**(i == 2) {

*insertionSort*(copy1c);

}

*time* = System.*nanoTime*() - *time*;

**switch** (i) {

**case** 0 :Insertion[0] = *compareCount*;

Insertion[1] = *moveCount*;

Insertion[2] = *time*;

**case** 1 :Insertion[3] = *compareCount*; // record details

Insertion[4] = *moveCount*;

Insertion[5] = *time*;

**case** 2 :Insertion[6] = *compareCount*;

Insertion[7] = *moveCount*;

Insertion[8] = *time*;

}

}

System.***out***.println("" + *array2String*(Insertion)); // print out numbers

*moveCount* = 0;

*compareCount* = 0;

String[] copy2a = arr.clone();

String[] copy2b = arr1.clone(); // clone arrays for use

String[] copy2c = arr2.clone();

**for**(**int** i = 0; i<3; i++) {

*time* = System.*nanoTime*();

*compareCount* = 0;

*moveCount* = 0;

**if**(i == 0) {

*shellSort*(copy2a);

}

**if**(i == 1) {

*shellSort*(copy2b); // go through each sort

}

**if**(i == 2) {

*shellSort*(copy2c);

}

*time* = System.*nanoTime*() - *time*;

**switch** (i) {

**case** 0 :Shell[0] = *compareCount*;

Shell[1] = *moveCount*;

Shell[2] = *time*;

**case** 1 :Shell[3] = *compareCount*; // record details

Shell[4] = *moveCount*;

Shell[5] = *time*;

**case** 2 :Shell[6] = *compareCount*;

Shell[7] = *moveCount*;

Shell[8] = *time*;

}

}

System.***out***.println("" + *array2String*(Shell)); // print out numbers

*moveCount* = 0;

*compareCount* = 0;

String[] copy3a = arr.clone();

String[] copy3b = arr1.clone(); // clone arrays for use

String[] copy3c = arr2.clone();

**for**(**int** i = 0; i<3; i++) {

*time* = System.*nanoTime*();

*compareCount* = 0;

*moveCount* = 0;

**if**(i == 0) {

*quickSort*(copy3a, **new** StringComparator());

}

**if**(i == 1) {

*quickSort*(copy3b, **new** StringComparator()); // go through each sort

}

**if**(i == 2) {

*quickSort*(copy3c, **new** StringComparator());

}

*time* = System.*nanoTime*() - *time*;

**switch** (i) {

**case** 0 :Quick[0] = *compareCount*;

Quick[1] = *moveCount*;

Quick[2] = *time*;

**case** 1 :Quick[3] = *compareCount*; // record details

Quick[4] = *moveCount*;

Quick[5] = *time*;

**case** 2 :Quick[6] = *compareCount*;

Quick[7] = *moveCount*;

Quick[8] = *time*;

}

}

System.***out***.println("" + *array2String*(Quick)); // print out numbers

*averageInsertComp* = Insertion[0] + Insertion[3] + Insertion[6];

*averageInsertMove* = Insertion[1] + Insertion[4] + Insertion[7];

*averageInsertTime* = Insertion[2] + Insertion[5] + Insertion[8];

*averageShellComp* = Shell[0] + Shell[3] + Shell[6];

*averageShellMove* = Shell[1] + Shell[4] + Shell[7];//average out the operations and time

*averageShellTime* = Shell[2] + Shell[5] + Shell[8];

*averageQuickComp* = Quick[0] + Quick[3] + Quick[6];

*averageQuickMove* = Quick[1] + Quick[4] + Quick[7];

*averageQuickTime* = Quick[2] + Quick[5] + Quick[8];

System.***out***.println("Averages for Insertion\n" + *averageInsertComp* +" "+ *averageInsertMove* +" "+ *averageInsertTime*);

System.***out***.println("Averages for Shell\n" + *averageShellComp* +" "+ *averageShellMove* +" "+ *averageShellTime*);

System.***out***.println("Averages for Quick\n" + *averageQuickComp* +" "+ *averageQuickMove* +" "+ *averageQuickTime*);

*moveCount* = 0;

*compareCount* = 0;

System.*exit*(0);

}

Details of Experiments and Results

First of all to note is the scope of arrays with which I would observe. I decided to manufacture my code so that I would have 3 arrays of the same size but diverging on their elements and structure. These arrays would then each be cloned and passed into each of the sorting algorithms where their practical attributes would be logged. This would give me plenty of reliably data and averages on each function for that particular size. Keeping with the theme of 3, I decided to pick that same amount of different sizes for analysis. I landed on the sizes 5000, 50000 and 100000. The code is some what flexible in that you can enter whatever size you wish to be analysed but because of the O(n^2) nature of Insertion sort, run times greatly inflated very quickly.

I will not lay out the empirical data gathered in tables, note that I am converting time from nanoseconds in the code to seconds.

Insertion: 5000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 6235846 | 6230847 | 0.075360383 |
| Arr2 | 6221560 | 6216561 | 0.063172425 |
| Arr3 | 6246103 | 6241104 | 0.048522259 |
| Average | 6234503 | 6229504 | 0.062351689 |

Shell: 5000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 97904 | 42906 | 0.002000427 |
| Arr2 | 101577 | 46579 | 0.003021031 |
| Arr3 | 103595 | 48597 | 0.002658552 |
| Average | 101025 | 46027 | 0.00256000333 |

Quick: 5000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 81587 | 13208 | 0.009051009 |
| Arr2 | 77984 | 13293 | 0.002125407 |
| Arr3 | 81394 | 13086 | 0.002437667 |
| Average | 80321 | 13195 | 0.00453802766 |

* Although this is a relatively small array we can clearly see the difference that shell sort has compared to the other algorithms.
* Two points of contention are the fact that shell sort is both faster and more consistent than its competitors.

Insertion:50000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 6.23096192x10^8 | 6.23046193x10^8 | 13.005217422 |
| Arr2 | 6.22129157x10^8 | 6.22079158x10^8 | 12.563588143 |
| Arr3 | 6.24744933x10^8 | 6.24694934x10^8 | 13.423677162 |
| Average | 6.233234273x10^8 | 6.2327342833x10^8 | 12.9974942423 |

Shell:50000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 1396737 | 696740 | 0.049384278 |
| Arr2 | 1452586 | 752589 | 0.046246574 |
| Arr3 | 1457262 | 757265 | 0.051758511 |
| Average | 1435528 | 735531 | 0.04912978766 |

Quick:50000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 2707707 | 137371 | 0.95269506 |
| Arr2 | 2698447 | 137001 | 0.165296171 |
| Arr3 | 2736543 | 136382 | 0.156600088 |
| Average | 2714232 | 136918 | 0.139055255 |

* Now we can see the differences between the sorting functions come to light.
* Insertion is on a different scale with the amount of time taken and operations completed.
* Quick does seem to be considerably less stable considered alongside Shell sort.

Insertion:100000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 2.498647692x10^9 | 2.49854769x10^9 | 48.218382636 |
| Arr2 | 2.494526492x10^9 | 2.494426493x10^9 | 47.557429787 |
| Arr3 | 2.499911786x10^9 | 2.499811787x10^9 | 69.991267669 |
| Average | 2.4976953x10^9 | 2.497595324x10^9 | 55.255693364 |

Shell:100000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 3004333 | 1504337 | 0.05525569336 |
| Arr2 | 3094599 | 1594603 | 0.130308703 |
| Arr3 | 3088708 | 1588712 | 0.111244607 |
| Average | 3062546 | 1562550 | 0.113581711 |

Quick:100000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Comparisons | Moves | Time |
| Arr1 | 9417422 | 275928 | 0.524854311 |
| Arr2 | 9629335 | 275708 | 0.374875493 |
| Arr3 | 9562131 | 275437 | 0.230973474 |
| Average | 9536296 | 275691 | 0.376901092 |

* A small note was my code broke since I had my counter variables initialised to int.

Final Conclusions and Comparisons with Theory

From my experiments I can conclude that:

* Insertion sort is incredible slow and has an insane amount of comparisons.
* Shell sort performs the best in both regards.
* Quick sort performs slightly worse than shell.

When looking back over the theory I can imagine that my implementation and testing of insertion sort was probably fine. The algorithm is extremely bad and scales terribly in both the theory and in practice.

The implementation of shell sort seems to be fine. The complexity of it is quite low at NlogN and this seems apparent in the testing.

Quicksort on the other hand seems to be badly implemented / tested. For example the pivot value is not averaged out and the input arrays may indeed be causing some trouble as the values can duplicate and I am aware that quick sort performs badly with duplicate values.

All in all though the results I extracted were not unexpected to any great degree. In future attempts at recording algorithm analysis I intend to spend more time on understand different quirks in java that would impede the testing process.

Peter Conway 15498698