[ALGORITHMS & EFFICIENCY]

Marcus Francis TIPLER,
Computer Science at Cardiff University

Marcus Tipler

Cardiff University | ComSC – Abacws Building

GOALS

Build an event driven object-oriented Java program that takes as input an unsorted list of randomly generated integers, implements basic structures and uses algorithms to benchmark different sorting algorithms on the arrays. We will then proceed to compare their respective performances and provide evidence as to which of our tested sorting algorithm is best.

MY APPROACH

Firstly, I decided to compartmentalise individual sorting algorithms in to their own classes for future ease of use.
For the ‘Quick Sort’ algorithm I decided to use a stack-based approach, due to tests instantiating that this method drastically improves speeds and uses less memory.
For the ‘Hybrid’ sorting classes, I used subarray lists of the size of a given threshold. This means that the sub arrays are only the size of the threshold and can be sorted by themselves before parsing them back in to their primary arrays to be sorted again.
I used a combination of ‘Quick Sort’ mixed with either ‘Selection Sort’ or ‘Insertion Sort’ where ‘Quick Sort’ sorts the already sorted subarrays.
All of this was planned, and mapped using Kanban and Infinite Canvas to keep track of progress and write out algorithms in Pseudocode before programming them in (find below).

EXTRACTS

```java
// ------------------------------------------------------------------
// Declares values for quick sort and reversed quick sort methods.
// ------------------------------------------------------------------
public int[] quickSort(List<Integer> Array, int elements) {
    int[] arr = new int[elements];
    for (int element = 0; element < elements; element++) {
        arr[element] = Array.get(element);
    }
    if(arr.length <= 0) return arr;
    Stack<Integer> stack = new Stack<>();

    stack.push(item:0);
    stack.push(arr.length - 1);
    while(!stack.isEmpty()){
        int high = stack.pop();
        int low = stack.pop();

        int pivotIdx = partition(arr, low, high);
        if(pivotIdx > low) {
            stack.push(low);
            stack.push(pivotIdx - 1);
        }
        if(pivotIdx < high && pivotIdx >= 0){
            stack.push(pivotIdx + 1);
            stack.push(high);
        }
    }
    return arr;
}
```

‘Quicksort Algorithm’

```java
private int partition(int[] arr, int low, int high){
    if(arr.length <= 0) return -1;
    if(low >= high) return -1;
    int l = low;
    int r = high;

    int pivot = arr[l];
    while(l < r){
        while(l < r && arr[r] >= pivot){
            r--;
        }
        arr[l] = arr[r];
        while(l < r && arr[l] <= pivot){
            l++;
        }
        arr[r] = arr[l];
    }
    arr[l] = pivot;
    return l;
}
```

‘Quicksort Algorithm Partition’

```java
public int[] insertionSort(List<Integer> Array, int elements){

    int[] arr = new int[elements];
    for (int element = 0; element < elements; element++) {
        arr[element] = Array.get(element);
    }
    if(arr.length <= 0) return arr;

    for( int i=0; i<arr.length-1; i++ ) {
        for( int j=i+1; j>0; j-- ) {
            if( arr[j-1] <= arr[j] )
                break;
            int temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
        }
    }
    return arr;
}
```

'Insertion-Sort Algorithm'

```java
public class sortSelection {
    public int[] selectionSort(List<Integer> Array, int elements){
        int[] arr = new int[elements];
        for (int element = 0; element < elements; element++) {
            arr[element] = Array.get(element);
        }
        if(arr.length <= 0) return arr;
        for(int i = 0; i < arr.length-1; i++){
            int min = i;
            for(int j = i+1; j < arr.length; j++){
                if(arr[j] < arr[min]){
                    min = j;
                }
            }
            if(min != i){
                int temp = arr[min];
                arr[min] = arr[i];
                arr[i] = temp;
            }
        }
        return arr;
    }
```

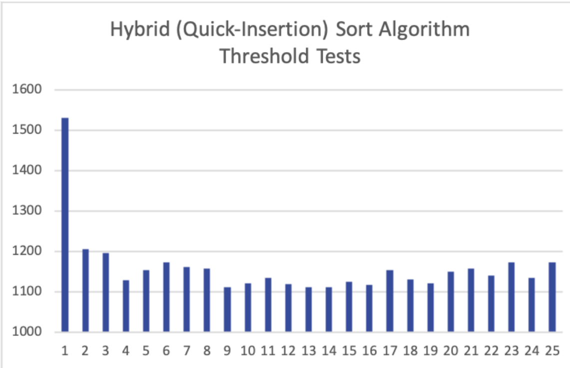
'Selection-Sort Algorithm'

```java
public int[] hybridSortQI(List<Integer> Array, int elements, int
threshold) {
    int[] arr = new int[elements];
    List<List<Integer>> subList = new ArrayList<>();
    List<Integer> ArrayCopy = new ArrayList<>(Array);
    for (int element = 0; element < elements; element++) {
        arr[element] = Array.get(element);
    }
    if(arr.length <= 0) return arr;

    for (int i = 0; i < elements; i += threshold) {
        subList.add(Array.subList(i, Math.min(i + threshold,
elements)));
    }

    int index = 0;
    for (int i = 0; i < subList.size(); i++) {
        List<Integer> cache = new ArrayList<>(subList.get(i));
        // Sort the subarrays using Insertion or Selection sort.
        int[] cacheSorted = sI.insertionSort(cache, cache.size());

        for (int j = 0; j < cacheSorted.length; j++) {
            ArrayCopy.set(index++, cacheSorted[j]);
        }
    }
    // Sort the entire array using quicksort
    arr = sQ.quickSort(ArrayCopy, ArrayCopy.size());
    return arr;
}
```

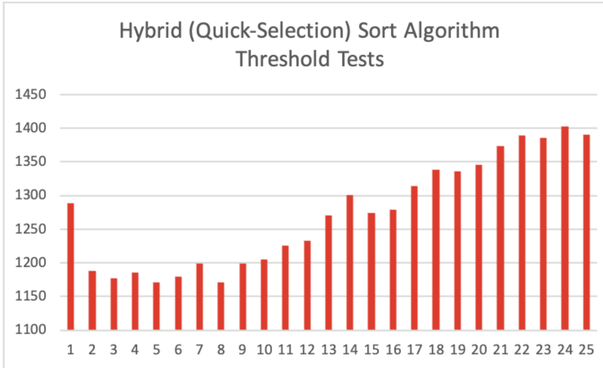
'Hybrid-Sorting Algorithm'

BENCHMARKING

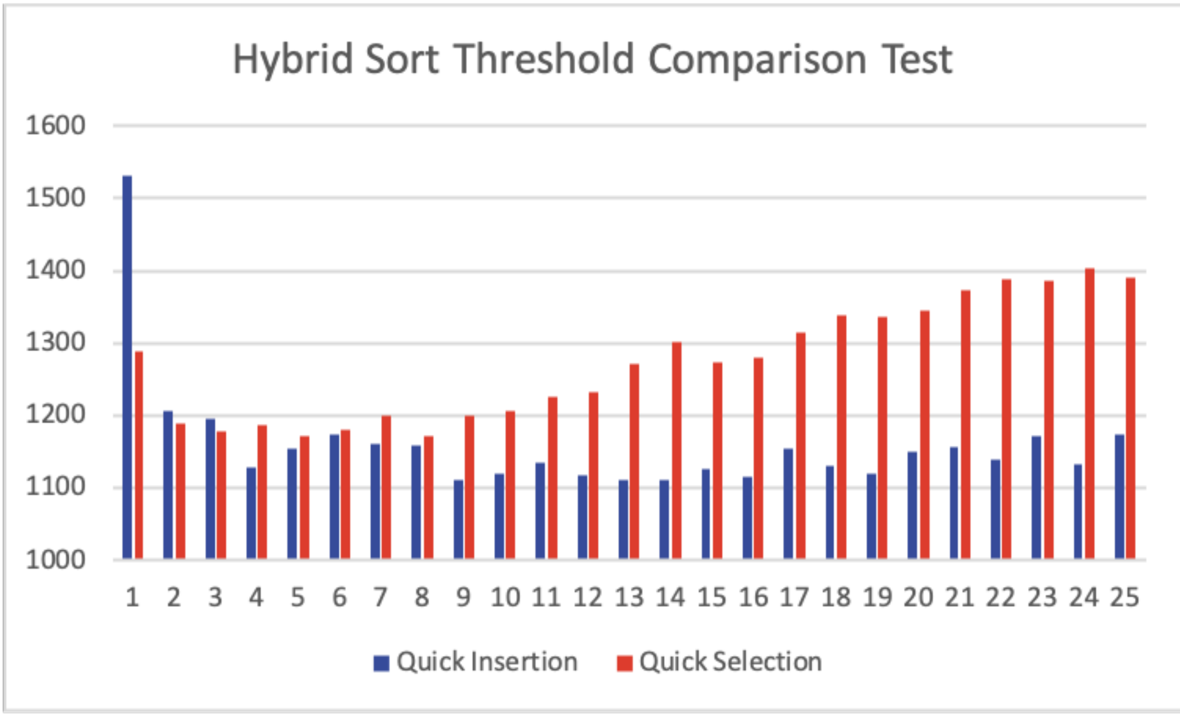
Firstly, to benchmark the hybrid sorting algorithms whilst at their peak efficiency, we needed to test which 'threshold' was best for each Hybrid Sort, so 'benchmarkComputerThreshold.java' was made, this file outputs a 'Comma Separated Values (CSV)' that is readable by Microsoft Excel. Once complete, benchmark performances can be graphed accordingly:



'Quick-Insertion' threshold tests
x: threshold value.
y: time in ms.



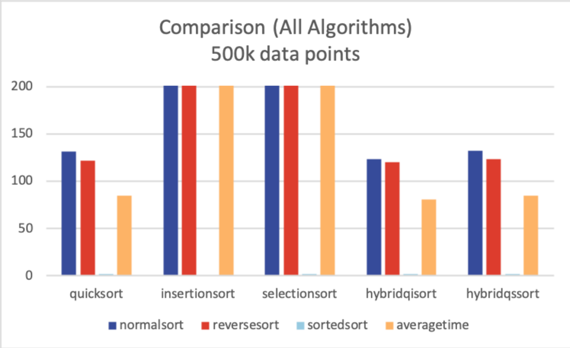
'Quick-Selection' threshold tests
x: threshold value.
y: time in ms.



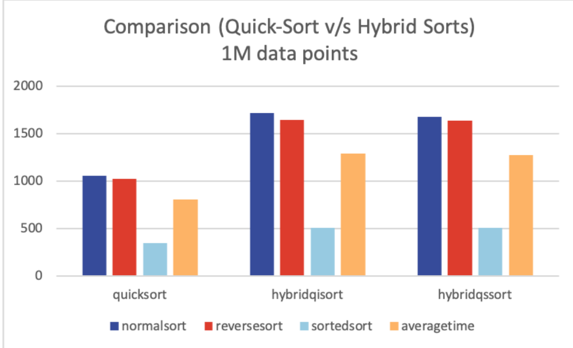
'Hybrid Sorts' compared
x: threshold value.
y: time in ms.

This shows that the 'Quick Insertion' sorting method has a better average performance at a threshold of 14 and that the most efficient 'Quick Selection' sorting can be is at a threshold of 8.

Now it’s possible to benchmark the algorithms themselves as follows:



All algorithms: compared (500k random values).



Algorithms of interest: compared (1M random values).

(all random values are using same seed for more accurate testing)

This shows that the performance of quicksort is outweighed by that of both Hybrid Sorts by small margins for an array size of 500k values, but the opposite effect occurs once the array size is increased to 1M random values.

EVIDENCE

(loading a text file with over 500k lines was impossible for my machines, please find attached ZIP archive with file called ‘sortedProof.txt’ for full output of the Quicksort algorithm).

```
Sorted list: 2 3 4 4 7 7 8 8 8 9 10 13 14 15 16 19 20 20 21 21 21 23 23 24 24 25 25 26 26 27 27 27 28 28 29 29 30 32 32 32 33 34 34 36 36 36 39 39 41 42 43 45 46 46 47 47 48 50 51 52 52
53 54 55 57 57 58 58 62 62 62 65 66 66 66 66 67 69 69 69 69 71 72 72 73 76 76 77 79 80 81 82 84 85 85 86 86 88 88 89 90 91 91 92 93 94 94 94 96 98 98 99 100 100 100 100 102 103 105 109
109 110 112 113 113 114 115 115 117 117 117 120 120 123 123 123 124 125 125 126 128 128 129 130 130 131 132 132 132 132 132 132 133 134 135 135 135 137 137 139 141 141 141 142 143 145
145 149 150 151 151 151 153 154 156 156 157 157 159 159 162 162 162 163 165 167 167 170 170 171 172 173 173 175 177 178 180 180 181 181 181 184 184 185 186 186 186 186 186 188 189 190
499916 499916 499917 499917 499917 499918 499919 499922 499923 499923 499925 499925 499926 499927 499930 499930 499932 499932 499933 499935 499935 499937 499938 499938 499938 499939
499939 499939 499940 499941 499941 499943 499944 499944 499947 499951 499952 499952 499954 499955 499955 499956 499959 499960 499962 499962 499963 499963 499964 499964 499965 499966
499968 499968 499969 499969 499970 499971 499971 499972 499972 499973 499974 499975 499975 499977 499978 499980 499980 499982 499982 499982 499983 499984 499984 499984 499986 499989
499990 499991 499992 499993 499994 499996 499996 499996 499998
Sorted list (in reverse): 499998 499996 499996 499996 499994 499993 499992 499991 499990 499989 499986 499984 499984 499984 499983 499982 499982 499982 499980 499980 499978 499977
499975 499975 499974 499973 499972 499972 499971 499971 499970 499969 499969 499968 499968 499966 499965 499964 499964 499963 499963 499962 499962 499960 499959 499956 499955 499955
499954 499952 499952 499951 499947 499944 499944 499943 499941 499941 499940 499939 499939 499939 499938 499938 499938 499937 499935 499935 499933 499932 499932 499930 499930 499927
499926 499925 499925 499923 499923 499922 499919 499918 499917 499917 499917 499916 499916 499916 499915 499914 499912 499910 499910 499909 499908 499908 499906 499905 499902 499902
157 156 156 154 153 151 151 151 150 149 145 145 143 142 141 141 141 139 137 137 135 135 135 134 133 132 132 132 132 132 132 131 130 130 129 128 128 126 125 125 124 123 123 123 120 120
117 117 117 115 115 114 113 113 112 110 109 109 105 103 102 100 100 100 100 99 98 98 96 94 94 94 93 92 91 91 90 89 88 88 86 86 85 85 84 82 81 80 79 77 76 76 73 72 72 71 69 69 69 69 67
66 66 66 66 65 62 62 62 58 58 57 57 55 54 53 52 52 51 50 48 47 47 46 46 45 43 42 41 39 39 36 36 36 34 34 33 32 32 32 30 29 29 28 28 27 27 27 26 26 25 25 24 24 23 23 21 21 21 20 20 19 16
15 14 13 10 9 8 8 8 7 7 4 4 3 2
```

This proves the algorithm is indeed sorting all of the values as requested.

CONCLUSION

Overall, from what it seems thanks to the data the program provided, the Quick Sort function is much more efficient for larger datasets, although slower for the given data we were originally required to test.

>> Pass : 1
>> Amount of Items : 1000
>> Time to Sort : 1.2s
>>

Benchmark ?

• Array (Randomly generated.)
↓
one for 1K, 5K, 10K, 50K, 75K, 100K and 500K items

• Apply the quicksort algorithm to the arrays

Focus #1 :

AmountofItems = [1k, 5k, ... 500k]
loop for range of elements in array Amount of Items.
week = ArrayCreator (amountOfItems[loop])
end loop
loop for range Amount of Items
weekSorted = algorithmQuickSort (week [loop])
end of loop
millisEndCapture()

for loop then save to variable. (list)

Use Millis capture before Start
millisCapture()

Do three captures and Compute average time.

Make function for Average Calculation (for 3 runs)

? Alternative to np.mean ?

Create Array
>> [1000 , 999 , 998 ..., 0]

What's needed ?
↳ A method that takes a list of item Amounts
outputs a list of Random Integers (lowest = 1 Highest = amount of items) * create items

For Sort Algorithms
is Sorted already
↳ pass
↳ if not
↳ Sort them

Problem method found.

Hoare partition scheme
List<integer> listItems
int pivot = listItems[0] # first array position
int Low = 0 - 1
int High = listItems.len() + 1
While True
↳ Low += 1
while listItems [low] < pivot
↳ Low += 1
High -= 1
while listItems [High] > pivot
↳ High -= 1
if Low >= High
↳ return High
listItems[Low], listItems[High] = listItems[High], listItems[Low].

Array(Integer) database // 1K, 5K, 10K, 25K, 50K, 100K, 500K. Parallel arrays
clockStart ();
sort QuickNormal ();
clockNormal (); // Use as clock Start for next Sort execution
sort Quick Reverse ();
Clock Reverse ();
Sort QuickSorted ();
Clock Sorted (); // finish Realtime clock.
Calculate AverageTimer();

Save values Accessible to clock variable for later.

↳ Inside clock method: print necessary Statements

↳ Calculate time for all values
>> Sorting Algorithm : 1 (-)
>> ~~Average~~ execution time = X ms
>> Sorting
>>
>>
>>
>> Average program execution time : X ms
perhaps mention what Sorting Algorithm is used ?

Week values Saved in array
Scale - able
All DATA { Array of item Amounts
All 3 Sorted Arraylist, 1 unsorted.
All clock initializations.

for amount Items . length Somehow
↳ Array[list]. append (n.f. Create Array (Amount Items [i]))

clock init
for Array[list]. size()
↳ quicksort (Array[list]. get(i)

Causes unexpected ERROR.
Sorted Array does NOT function in sort Algorithm.
↳ Debug with Array of 1, 1, 1, 1, 1, 1, 1.
NEEDS FIX ⚠
⚠ This works but other Sorted Arrays do not! ⚠
↳ needs research or Alternative debugging.

THE PROGRAM DEBUGGING NIGHTMARE.

How I plan to Solve:

1. Compare Solution to other Solutions, even in other languages (python or c..)
↳ Debugger is proving to be useless due to large dataset and intermittency of the fault
2. Stack overflow / Reddit for possible answers -> an Stackoverflow query = plan #030
(plan #0)
3. Manually stepping functional program, attempting to analyze the possible faults or generally something missing. (open)
(not there)

• Could be a problem w/ the recursive code. -> try creating a temporary class that isn't recursive.
↳ Could also be much more efficient!
① problem is: I have ALREADY DEVELOPED a recursion method.
↳ ~~Possibly Unfixable ??~~

⚠ Stackoverflow ISSUE w/ RECURSION
↳ Function is to be RE-DEVELOPED for larger datasets.
in it's worst case scenario.

100k limit but much changing for that ⚠

ISSUE SEEMINGLY FIXED
NEW ISSUES TO ARISE.
↳ Rename to LONGER functions.
↳ TIME DELAY is HIGH!
SB ? Small

TESTING NEW VALUE CHANGE IN CODE DID NOT WORK.. :C

COMPILING WAS OF A GOOD IDEA! (Rummel)

Need Sorted list prior to Benchmarking.
↳ Run Quicksort Algorithm and create Sorted version of "week" variable.

week { day { 0, 1, 2, 5, 9, 7 }
day { ____ ____ }

weekSorted { daySorted { __ __ __ }
daySorted { __ __ ____ }
✓

Step back : What am I missing?

Requirements : Use QuickSort until list is small enough.
↓
Start insertion Sort until complete.
Need fully functional insertion Sort.
By itself.

Need Monitored threshold that can be changed for Benchmarking.

Algorithm Benchmark . Java
Merge Sort . Java
Insertion Sort . Java
Quick Sort . Java

To Do :

- [x] Insertion Sort.
- [x] Testing runtime for Insertion Sort. } ✓ REQUIRES KNOWLEDGE OF 'PACKAGES'?
- [ ] Hybrid Sort.
- [ ] Testing runtime for Hybrid Sort.

Idea : turn in to CSV readable file and use Numpy w/ python ?

or use alternative. lol.

Where I left off:
Insertion Sort works
Just have to find break point to place threshold for Hybrid Sort. that's that then.

Somewhere in the Hybrid Sort I need to place a break point. IDK where. will look it up.
Also, algorithm Benchmark file already has import for Hybrid Sort but it's acting weird.
TB Checked !!!

How Merge Sort works

3 4 9 7 2 3
↓
insertion Sort

Sort Merge
// 20 elements : diminishing returns
Split arrays until elements in array = threshold.
insertionSort every list of size threshold
Append all array content to array.
Quick Sort the entire array.

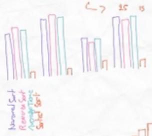
for Array list long i + threshold
ArrayList . Sublist <= ArrayList<ArrayList<INTEGER>>

CSV file
Title, NormalSort, ReverseSort, SortedSort, AverageTime

1K 5K 10K 50K 75K 100K 500K

- [ ] What's the best Sorting Algorithm ?
- [ ] What's the best Sorting Algorithm (more detailed)?
- [ ] What's the best threshold? 5 - 25
↳ 25 is 5 more than 20 is returns.

Diminishing Returns.



Should look like this

Best Score .