Data Structures and Algorithms

100166648 - SSQ16SHU

20/03/19

Exercise 1

Part 1.1: Informal Description of Algorithm

the idea underlying this approach to the SVD problem was a nested for-loop.
For each of the elements in the array, i looped through it counting the times that value appeared. if the count was greater than the previous highest count, the value was replaced with the new potential SVD.
If an SVD WAS found, its value was printed to screen.
If the highest count of the potential-SVD was not greater than half the size of the array+1 (to make it the majority for odd or even sized arrays) then a failure to find SVD was printed to screen in its place.

Part 1.2: Formal Description of Algorithm

Algorithm 1 svdP1(**T**,n)

Require: Array **T** of length n
Ensure: print of SVD value or failure to find SVD notice

```
1: potentialSVD                          ▷ temp holder for highest-frequency value
2: svdCount                              ▷ counter for current highest-frequency
3: for i ← 1 to n do
4:     contenderCount = 0                ▷ current element frequency-counter
5:     for j ← 1 to n do
6:         if T[j] == T[i] then
7:             contenderCount++
8:         end if
9:     end for
10:    if contenderCount > svdCount then
11:        svdCount = contnderCount
12:        svdVal = T[i]
13:    end if
14: end for
15: if svdCount > (n/2) + 1 then
16:    printsvdVal
17: end if
18: else: printfailnotification
```

Part 1.3: Run-time Complexity Analysis of Algorithm

Steps:

1. **Determine Fundamental Operation:**
   $T[j] == T[i]$

2. **Describe Case:**
   Worst case scenario

3. **Form the Run-time Complexity Function:**
   count fundamental operations

$$t(n) = \sum_{i=1}^{n}(\sum_{j=1}^{n} 1) \tag{1}$$

   Solving steps:
   if $\sum_{i=q}^{p} 1 = (p - q + 1)$
   then inner summation substituted as (n-1+1)

$$t(n) = \sum_{i=1}^{n}(n - (i-1) + 1) = \sum_{i=1}^{n}(n - i) \tag{2}$$

   and reduced further using $\sum_{i=q}^{p}(a + b) = \sum_{i=q}^{p} a + \sum_{i=q}^{p} b$
   to

$$t(n) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i \tag{3}$$

   converted using $\sum_{i=1}^{p} a = a \sum_{i=1}^{p} 1 = ap$

$$t(n) = (n \times n) - \sum_{i=1}^{n} i \tag{4}$$

   final conversion using the same rule

$$t(n) = (n \times n) - (i \times n) = n^2 - (n \times i) \tag{5}$$

4. **Characterize RT to give Order of the Algorithm:**
   run-time complexity of svdP1 is $t_{svdP1} = n^2$
   Algorithm is Order n^2, or $O(n^2)$
   This is therefore a quadratic algorithm

Part 1.4: Java Implementation of Algorithm

-see Zip archive submission of source file(s) 'DSaA_CW2'

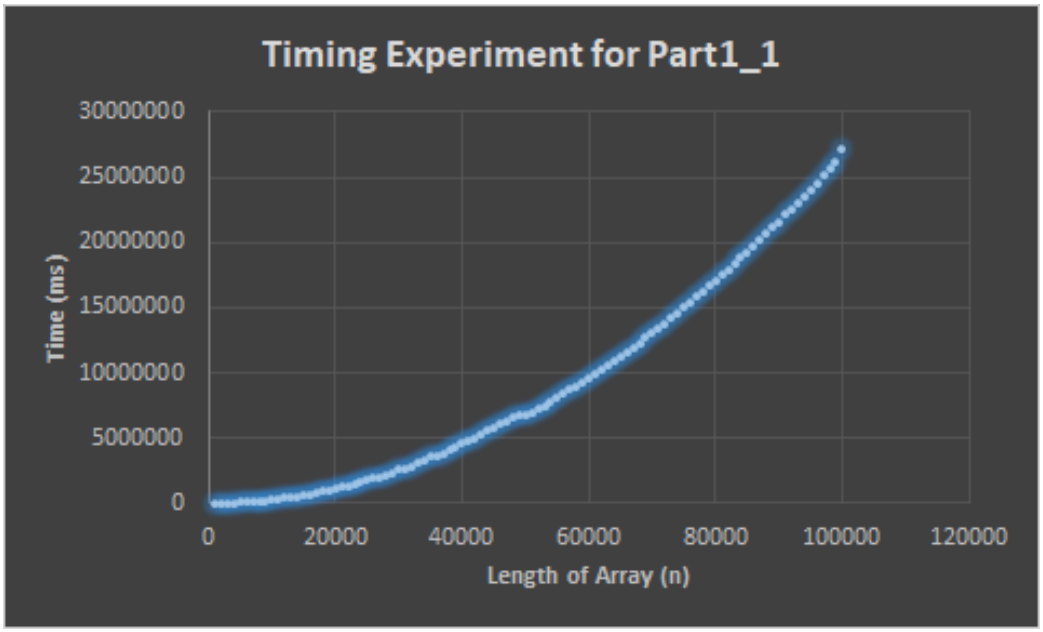
Part 1.5: Timing Experiments for part 1-1

For my timing experiments, I utilized existing classes i wrote previously (namely: Stopwatch.java and WriteFile.java, included in submitted source files). These files carried out the same function here as they originally did: noting the system's nanotime immediately before-and-after running the algorithm and recording the time elapsed between them.

These have been slightly modified however to reflect feedback received on DSaA CW1, modifying the repetitions and averages taken accordingly. Additional clases have been added for abstraction, containing the timing experiment methods and another to contain the algorithms themselves.
This allows the 'main' to simply run the program without becoming cluttered with unneeded test harness output

Their function of minimizing the impact of outliers through repetition and taking second averages of the dataset is preserved, though as mentioned slightly modified.
These now take the form of taking the average of 100 repetitions of the averages of 50 repetitions
(averages of a set recorded, repeated and averages of this set of first-averages recorded as final output)
Compensation for Java's inherent 'cold-start' has also been retained, discarding the first 10 complete sets before starting over for the recorded values to be taken.

Results of part1-1 Timing Experiments



Comparing results to initial analysis of the Run-time complexity, they confirm my analysis result of Order(n^2): Quadratic.
The workload illustrated on the above graph shows the increase of (n) work for every extra value of (n).
This experiment took so long to run that it can be safely concluded that this algorithm is practically useless for anything above the smallest values of (n).

Part 2.1: Informal algorithm description:

This algorithm version is based on the idea of using a SORTED array as input as well as using a simultaneous array to record the frequency of the elements in the array.
the array is traversed using a for-loop, the corresponding value in the frequency array is incremented if the next element matches the current.
when a match is not found, the array position value is moved to reflect the next element in the loop, setting the value to 1 (for the first found of the mis-matching element) and is subsequently compared against the next element and so on.

Part 2.2: Formal algorithm description:

Algorithm 2 dualArr(**T**,n)

Require: sorted array **T** of length n
Ensure: print of SVD if present, or failure message

1: sort T ▷ Using Javas .sort (Timsort variant)
2: $F[n]$ ▷ Array to hold frequency values
3: $freqPos \leftarrow 0$ ▷ position of element in F[]
4: $svdVal \leftarrow 0$ ▷ hold value for potential SVD
5: $svdCount \leftarrow 0$ ▷ holder for count of potential SVD
6: **for** $i \leftarrow 1$ to $n-1$ **do**
7: **if** $T[i] == T[i+1]$ **then** ▷ compare T element and next element
8: $F[freqPos]++$ ▷ increment freq counter element
9: **end if**
10: **Else:**
11: ▷ **check for new potential SVD**
12: **if** $F[freqPos] > svdCount$ **then**
13: $svdCount \leftarrow F[freqPos]$
14: $svdVal \leftarrow T[i]$
15: **end if**
16: **end for**
17: **if** $svdCount > n/2+1$ **then**
18: $printSVD$
19: **end if**
20: **Else:**
21: $printfailmessage$

Part 2.3: Run-time Complexity Analysis of Algorithm

Steps:

1. **Determine Fundamental Operation:**
   if $T[i] == T[i+1]$

2. **Describe Case:**
   Worst case scenario

3. **Form the Run-time Complexity Function:**
   Solving steps: The algorithm utilizes Java's sort function (which the documentation specifies the worst case complexity as: $O(nlogn)$) and proceeds to make a single pass through the array ($O(n)$)
   This results in $O(nlogn) + n$
   This simplifies down to remain as $O(nlogn)$

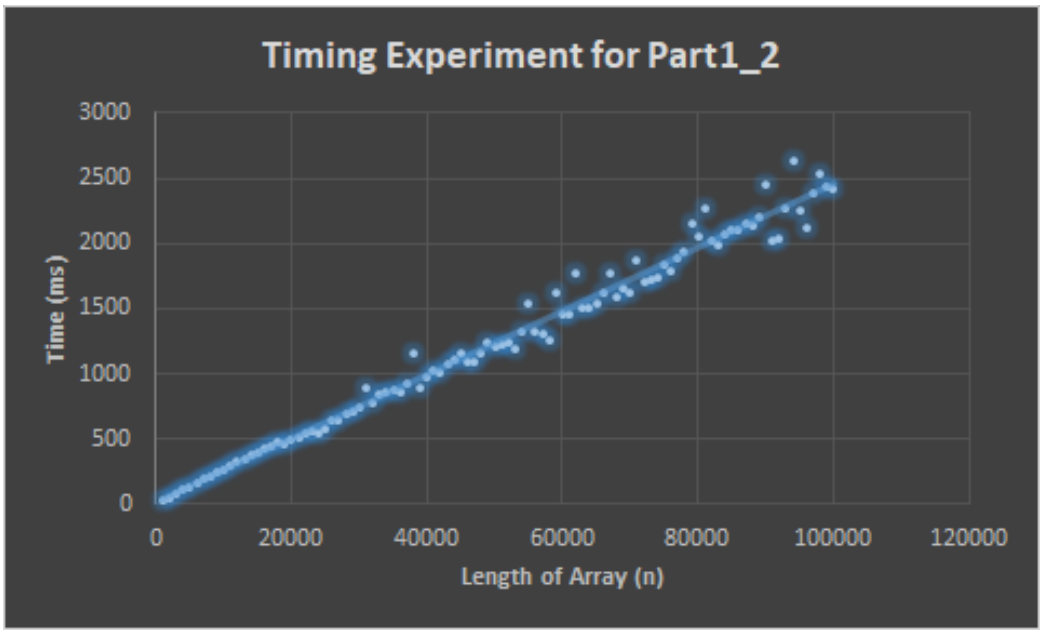
4. **Characterize RT to give Order of the Algorithm:**
   See above. This algorithm is characterized to be $O(n\ log(n))$

Part 2.4: Java Implementation of Algorithm

-see Zip archive submission of source file(s) (as above)

Part 2.5: Timing Experiments

For these experiments, i used the same method as outlined above in Part 1.4.
As the graph shows, there is an immeasurable improvement in the run time of these timing experiments. (From hours to minutes)
There are some outliers remaining in these results. The cause was attempted to be located through repetitions of the entire experiment, but this showed no obvious pattern to suggest a cause. Factors beyond my control (such as machine/network load) can be considered as potential causes, but this is not confirmed.



The above graph has had a linear trend line inserted to illustrate that the results have come close to, but not reached the optimal runtime solution ($O(n)$) for this problem.

Part 3.1: Informal algorithm description:

This version of the SVD solution is an implementation of the Boyer-Moore Voting Algorithm.
This method requires no additional data structure storage (such as a hashmap), only requiring some variables (a counter and a pointer to the next comparison). The algorithm consists of two steps.

The first selects a candidate for the SVD by checking an element against the next in the array. If a match is found the counter is incremented. If a match is not found, the counter is decremented. this repeats until the counter reaches zero.
At which point, the current element being assessed is set as the final 'next-element' when the counter reached zero.
The process is then repeated from this point. Any previous candidate cannot be an SVD (or if it IS then it will be found again further along the array)

The second step is to once again pass through the array counting the times that the candidate appears in the array (SVD being found if the count is greater than half of the array length).
If no SVD is found, the method will return '0', to be used by any printing methods to output this as text.

Part 3.2: Formal algorithm description:

Algorithm 3 linearAlg(**T**,n)

Require: array **T** of length n
Ensure: print of SVD if present, or failure message

```
candidate                                   ▷ variable to hold candidate element
counter ← 1                                                        ▷ vote counter
index ← 0                                   ▷ potential candidate element index
i ← 1                                                              ▷ loop position
for i to n do                                                  ▷ loop through array
    if T[index] = T[i] then
        counter +1
    else
        counter -1
    end if
    if counter = 0 then
        index ← i
        counter ← 1
    end if
end for
candidate ← T[index]                                  ▷ assign candidate element
i ← 1                                          ▷ pass through array second time
tally ← 0                    ▷ number of candidate element found in array
for i to n do
    if T[i] = candidate then
        counter +1
    end if
end for
if tally > n/2 then                  ▷ check tally exceeds half array length
    return true
    else
    return false
end if
```

Part 3.3: Run-time Complexity Analysis of Algorithm

Steps:

1. **Determine Fundamental Operation:**
   if $T[index] == T[i]$

2. **Describe Case:**
   Worst case scenario

3. **Form the Run-time Complexity Function:**

   Solving steps: this algorithm uses two single passes through the array, each carrying out n work

   $$t(n) = \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} 1 \tag{6}$$

   this can be re-written as

   $$t(n) = n + n \tag{7}$$

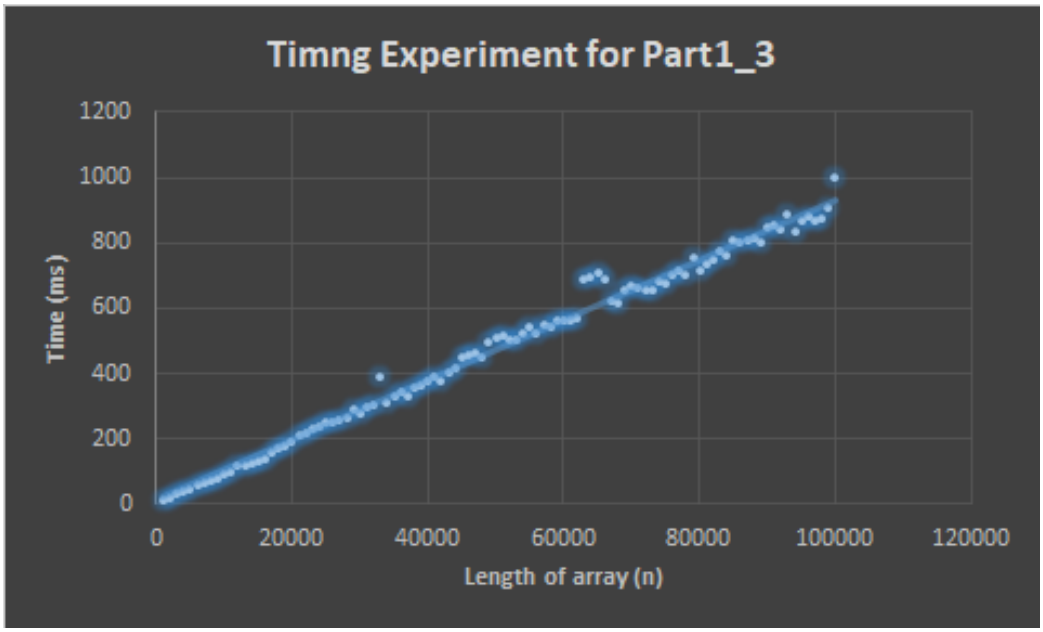
4. **Characterize RT to give Order of the Algorithm:**
   This algorithm is characterized to be linear $O(n)$

Part 3.4: Java Implementation of Algorithm

-see PASS submission of source file(s) (as above)

Part 3.5: Timing Experiments

For these experiments, i used the same method as outlined above in Part 1.4 and part 2.4.
This set of experiments however confirm my initial assessment of the algorithm's run-time complexity being linear.



Despite the attempts within the experiment and multiple repetitions of the entire experiment, some data does not strictly conform to the trend-line (inserted into graph to show this).
As seen in the results of Part 2, investigations into the cause of these pattern-less deviations were also inconclusive for this experiment.

Exercise 2

Parts 2.1-2.4

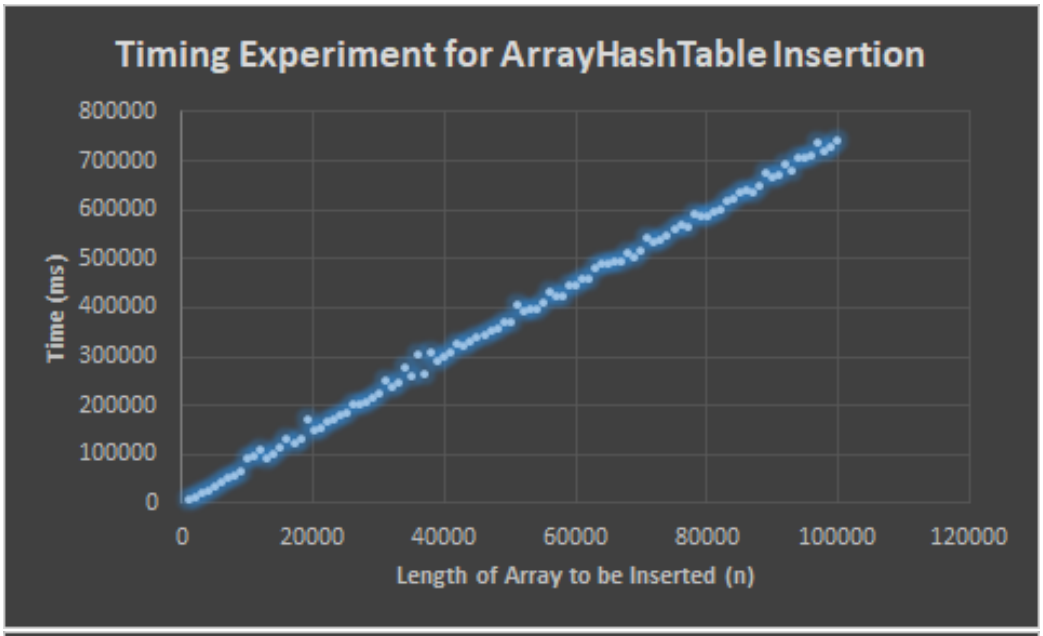
See submission of source files as above.

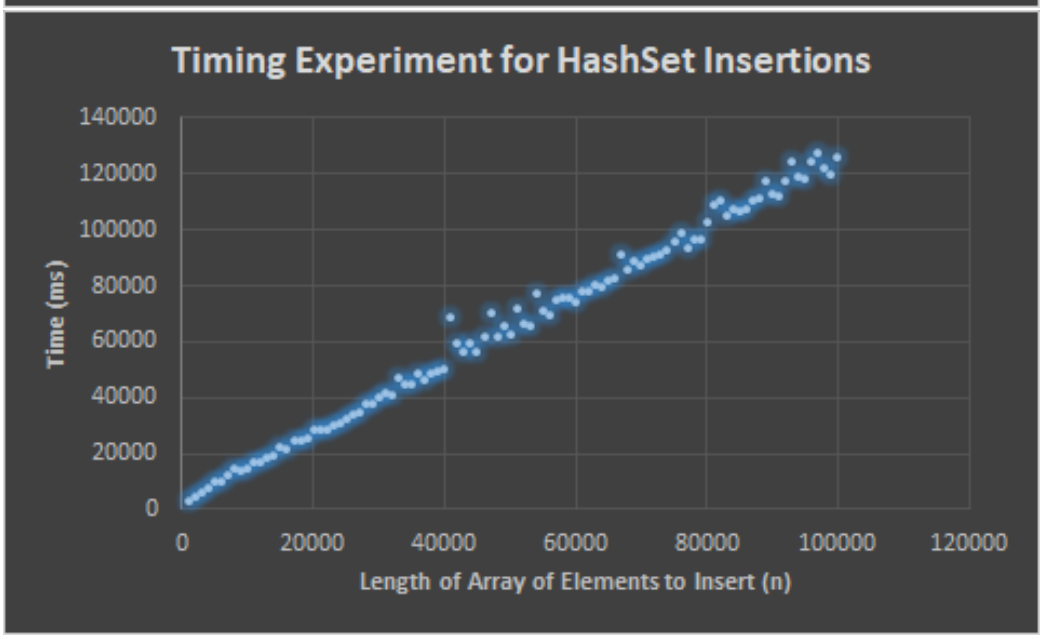
Part 2.5 Timing experiments

A modified version of the code for the timing experiments above was used for insertion and again for removal of (n) elements from both Java class HashSet and the ArrayHashTable class. The complete repetitions of the experiments (outer loop) had to be reduced by a factor of ten due to not being able to leave it running for hours while working at a public terminal.
These experiments can be defined as:
10*(50*(100 values of n))
as opposed to the above algorithm timing experiments of:
100*(50*(100 values of n)).

For clarification: Below I will present the output graphs of both classes for insertions, then offer my comparison.
This is then followed by graphs of their respective removal performances and further observations.

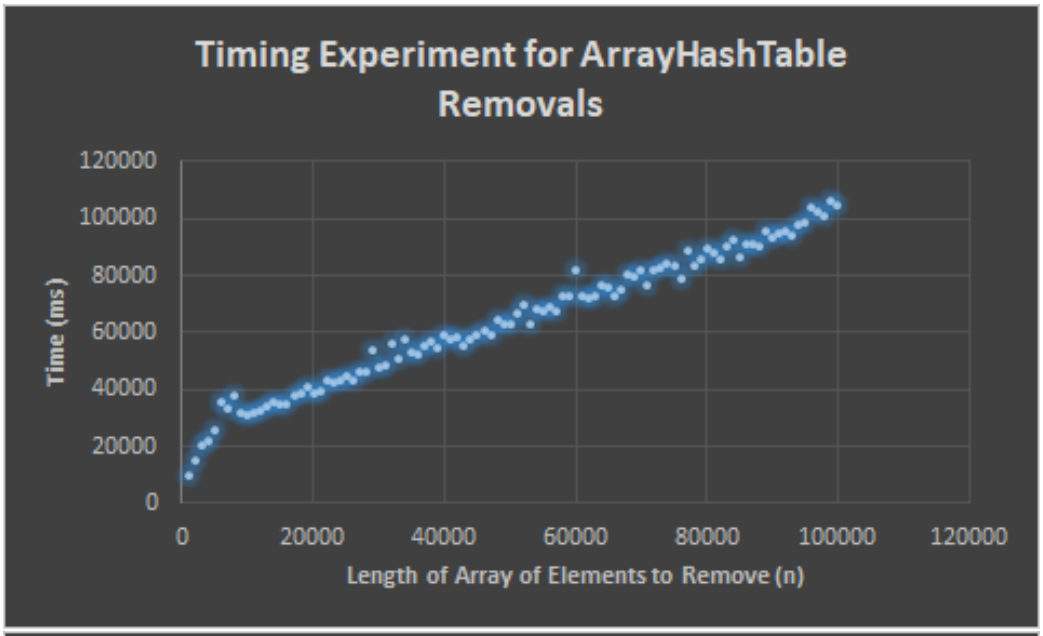
Part 2.5.1 Insertion Comparison

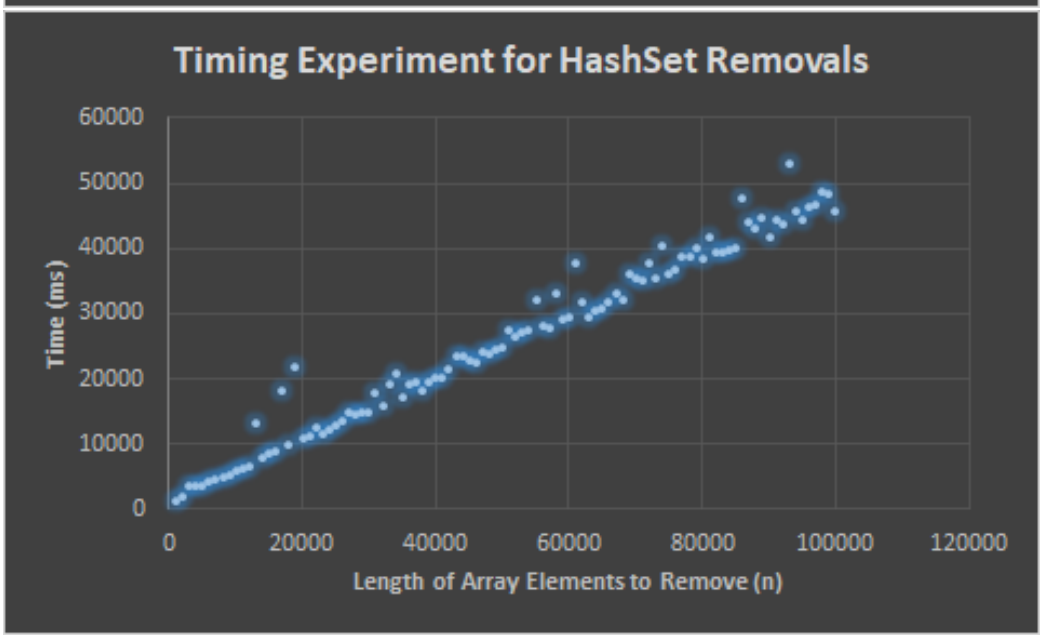




As can be seen above, The ArrayHashTable takes much longer than Java's HashSet for element insertion.
As using arrays as the data storage method for the bucket chaining is obviously not the preferred method, I was expecting this. The magnitude of difference did however surprise me a little (Approx. 5 times greater).
My main assumption of the cause for this is that of re-sizing the arrays. Whereas a linked list (or even an ArrayList) only needs to change a pointer to an additional node, the implementation required in the spec for ArrayHashTable is to double the array size upon exceeding its limit.
the additional time to create a new (longer) array and copy over each existing element to the new chain is very time-expensive.
This coupled with additional comparisons that would not be needed for another data structure (such as checking for null elements to prevent NullPointerExceptions when checking for an existing chain during addition of elements), further degrades the potential performance of this class.

Part 2.5.2 Removal Comparison





Again the HashSet can be seen to be faster on removal of elements, however the magnitude of difference is only approximately two greater, instead of five.
This supports my hypothesis that the main cause of delay during insertion is that of extending the arrays when they overflow.
Without this hindrance, the performance latency of ArrayHashTable is again mainly due to the methods outlined by the spec to work with an array-based data storage. (namely: my implementation of 'advance()')
This method attempts to mitigate some small delay by incorporating the act of 'removal' in its secondary function of moving the remaining elements in the array toward the start, filling the void left by the removed item.
This method does not actually deal with any removal, or setting of the element to null, etc. It simply begins the 'advance' from the next element after the removal target. Effectively removing it by ensuring the first 'advance' overwrites it.
Further functionality of reducing the size of the array upon removal if the removal caused the array to fall beneath half of its length was considered, but ultimately rejected due to the further degradation of performance this would have on the timing.(even without actually reducing it, the comparison after every removal would have a cumulative effect)
Another data structure such as a linked list would require much less work to remove an element, simply reassigning pointers to exclude the target element which would then be caught by the garbage collector.