An Oscillating Merge Sort implementation

Maximilian Mihaldinecz

Table of Contents

1 Part 1: Analysis of Comparison Sort Algorithms

1.1 Section A: Briefing on the selected sorting algorithms

Algorithm ***A*** is the simplest form of the pancake sorting (Dweighter, 1975) algorithms. It is considered to be a variant of the selection sort, which uses swaps instead of *flips* to move element V(i) in vector **V** to its final position. Flipping V(i) reverses the current sort order of all V(1..i) elements. A uses **n** maximum selections and at most 2n-1 flips to sort V, thus resulting in $O(n^2)$ time complexity for best, average, and worst cases. A is also an in-place and unstable algorithm.
Therefore, the initial performance expectation for A is to closely match bubble sort on average, nearly sorted, and reverse sorted arrays. However, bubble sort should outperform both algorithms on pre-sorted arrays.

Algorithm ***B*** is a basic (Knuth, 1998) version of the non-in-place oscillating merge sort (Sobel, 1962). It is a variation, that sorts **N** elements using **T** (T: $(T-1)^6 < N \leq (T-1)^7$, and $T \geq 4$) number of virtual tapes represented as a sparse T×N matrix, thus giving an $O(N\lceil \log_7 N \rceil)$ space complexity. When $N<(T-1)^7$, then $D=(T-1)^7-N$ dummy elements added to N, and will be removed when sorting is completed. This results in the time complexity being independent of the initial order of the elements, and worst/average/best case scenarios are being equal. However, D impacts time complexity making it $O((N+D)\log(N+D))$. Also considering a significant constant **C**, the initial performance expectation is to outperform the other two algorithms only when D is relatively small.

Both algorithms implemented iteratively and assume no pro-order checks.

1.2 Section D: Test results

1.2.1 Using random arrays up to 10^4 lengths.

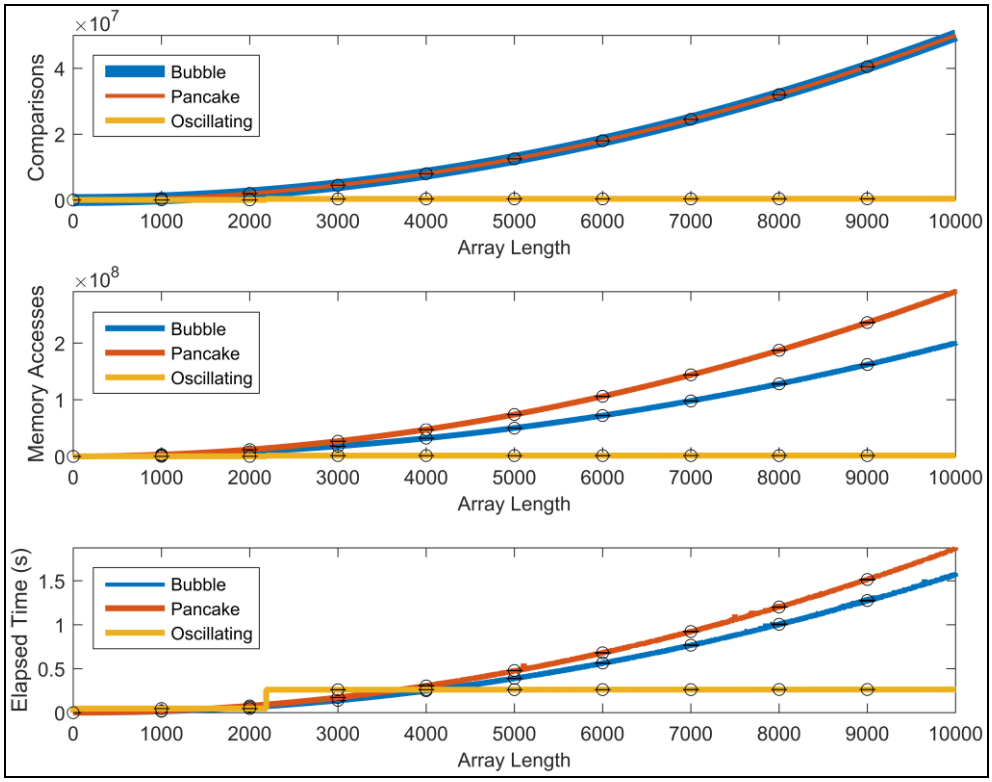

Figure 1 – Pancake and bubble sorts are executing nearly (but not exactly) the same amount of comparisons when using random arrays (pancake's line covers bubble's on the first plot). Pancake operates with higher amount of memory accesses than bubble, which is due to the numerous swaps during flips. As expected, oscillating shows a stair-like line at elapsed time as **N** requires the increase of **T**'s at N=2188 (3^7+1). Important to point out that oscillation uses significantly less memory accesses and comparisons than the other two algorithms. Measurements were taken of array lengths 2..10^4 in steps of 1.

1.2.2 Using nearly sorted (10%) arrays up to 3×10³ lengths

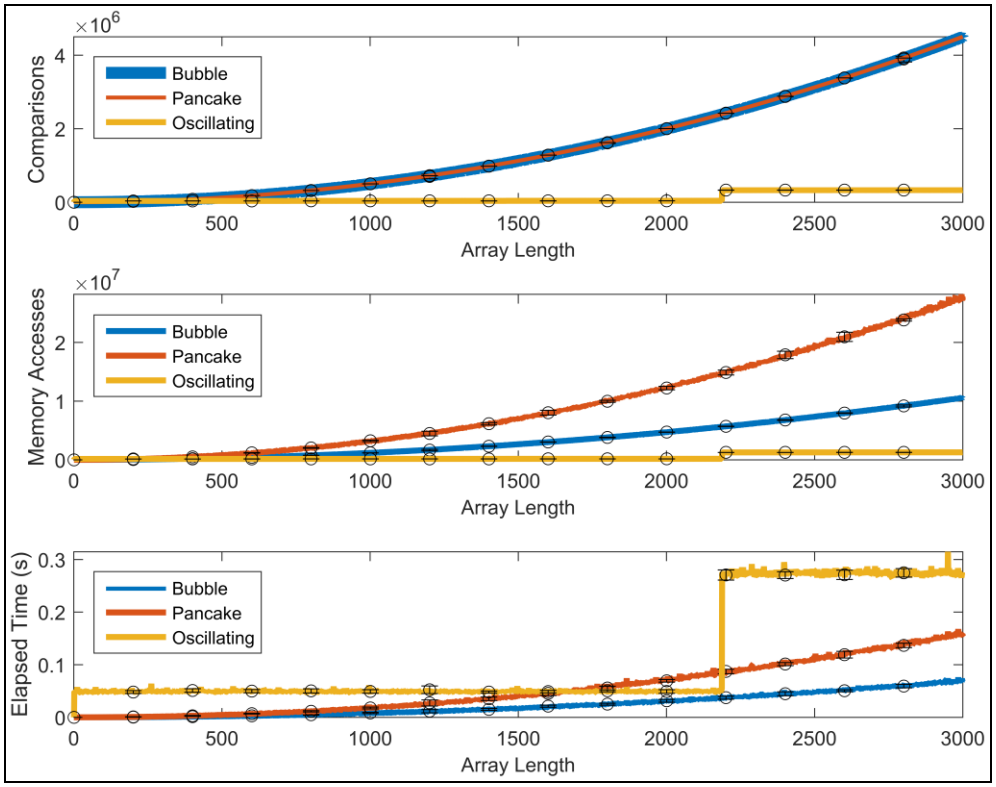

Figure 2 – Testing algorithms on sorted arrays where every 10th element has been replaced with a random value. Having the array nearly sorted lowered the memory access of the bubble sort, but had no other significant impact. Measurements were taken of array lengths 2..3×10³ in steps of 1.

1.2.3 Using nearly sorted (20%) arrays up to 3×10^3 lengths

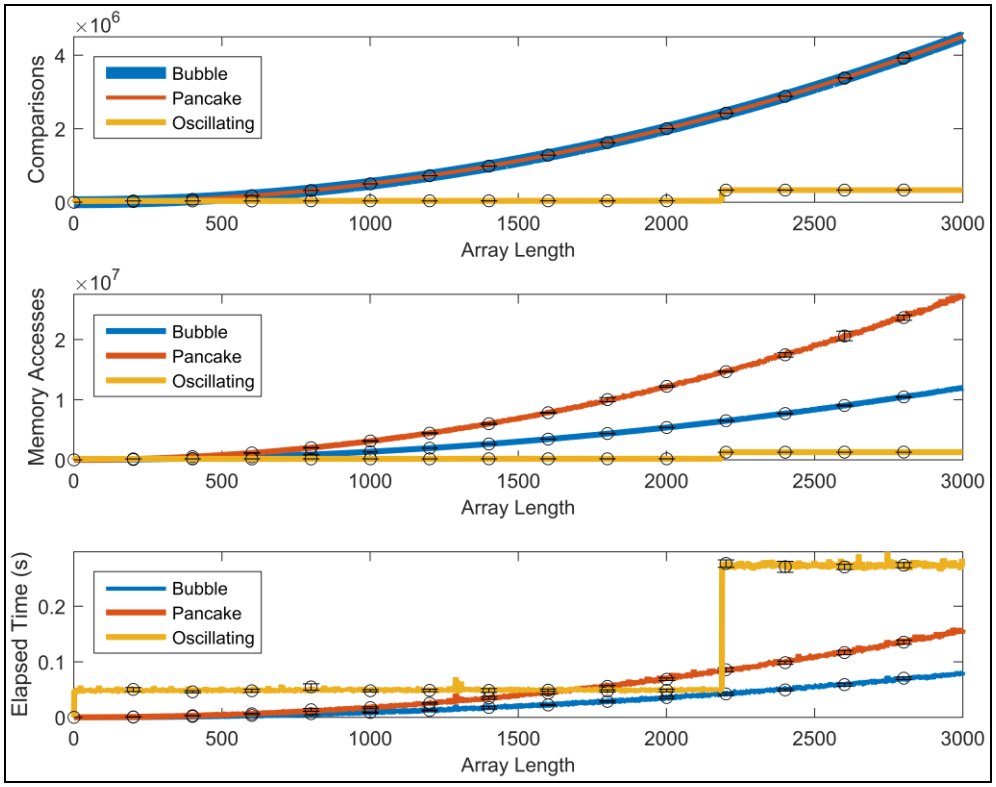

Figure 3 - Testing algorithms on sorted arrays where every 5th element has been replaced with a random value. Doubling the number of random elements compared to the previous figure made no cardinal difference. Measurements were taken of array lengths $2..3\times10^3$ in steps of 1.

1.2.4 Using reverse sorted arrays up to 3×10^3 lengths

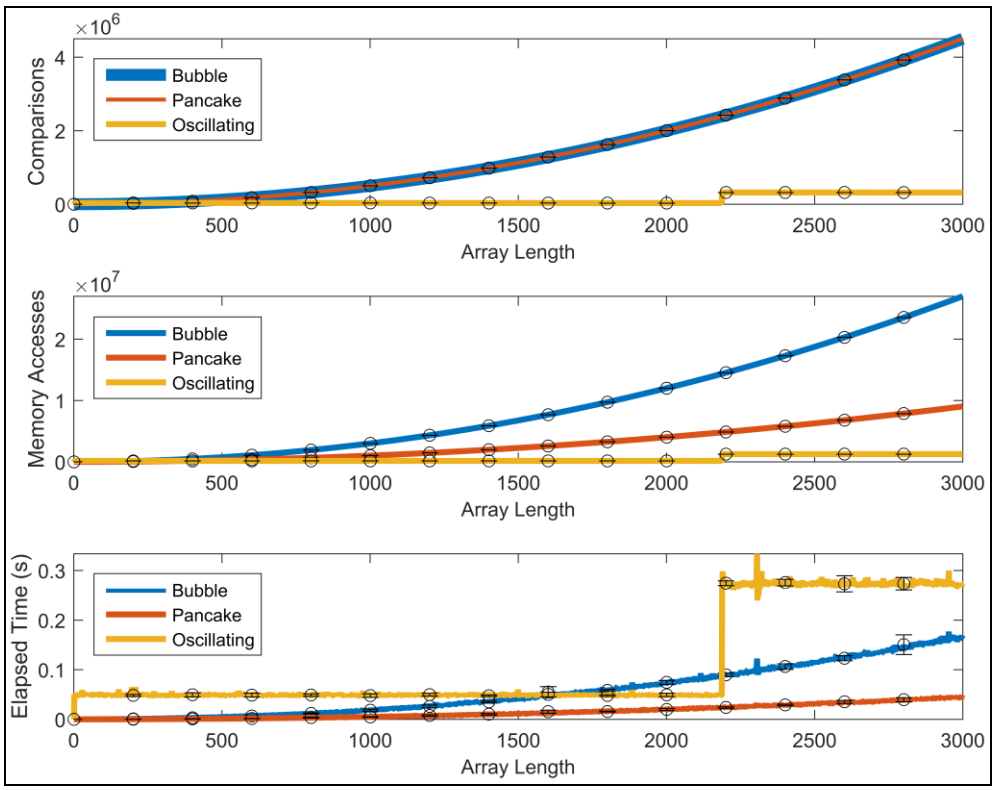

Figure 4 – Using reverse sorted arrays ensures that bubble will perform on its worst case. In this scenario, pancake was able to perform faster and with less memory access than bubble (related metrics). Pancake saves memory access as it now always has the next largest value at the beginning of the list, thus the first flip (of each flip pairs) will cost no swaps. The number of comparisons were not impacted by the reverse sorting. Measurements were taken of array lengths $2..3\times10^3$ in steps of 1.

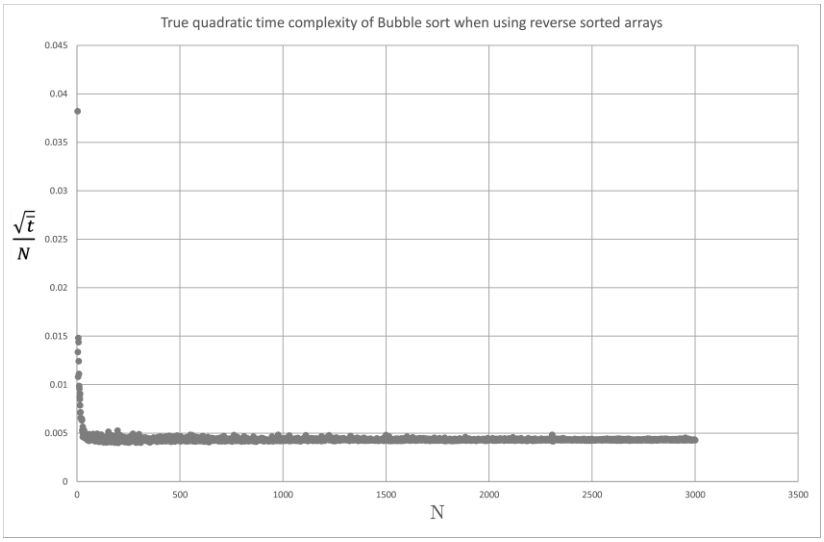

Figure 5 -Calculating the $\sqrt{\bar{t}}$ for bubble sort demonstrates its true quadratic time complexity for reverse sorted arrays.

1.2.5 Using pre-sorted arrays up to 3×10^3 lengths

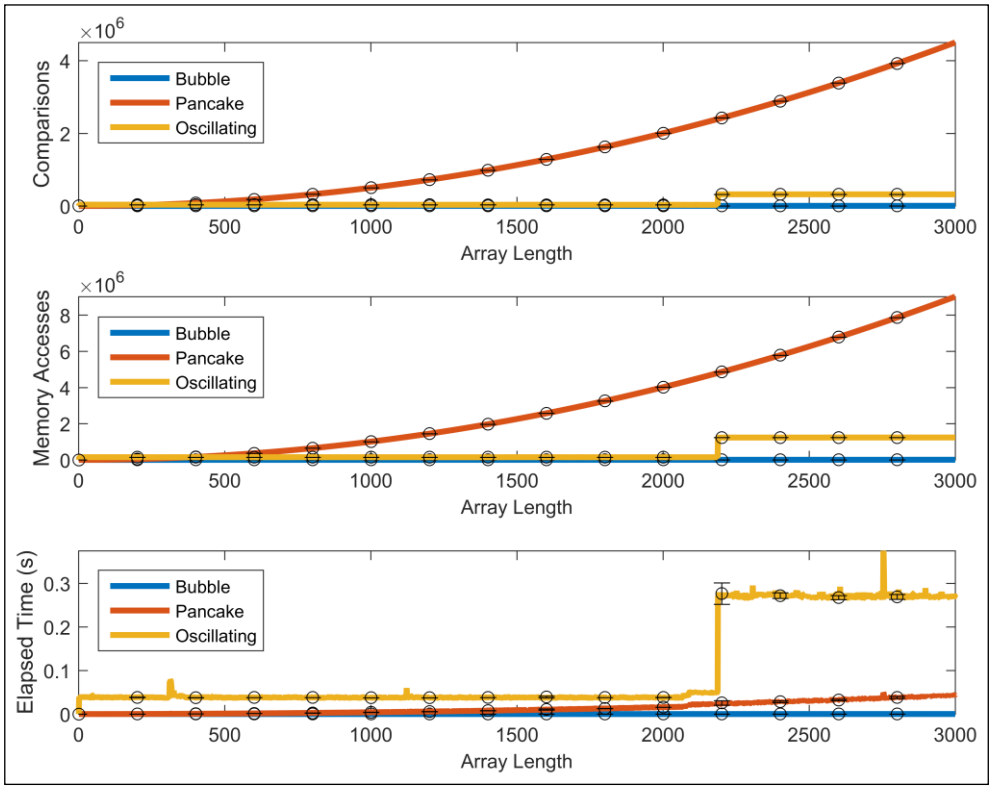

Figure 6 – Using pre-sorted array allows bubble to perform on its best (n) for all the three metrics. By not needing to flip, it also reduced the memory access for the pancake sort.

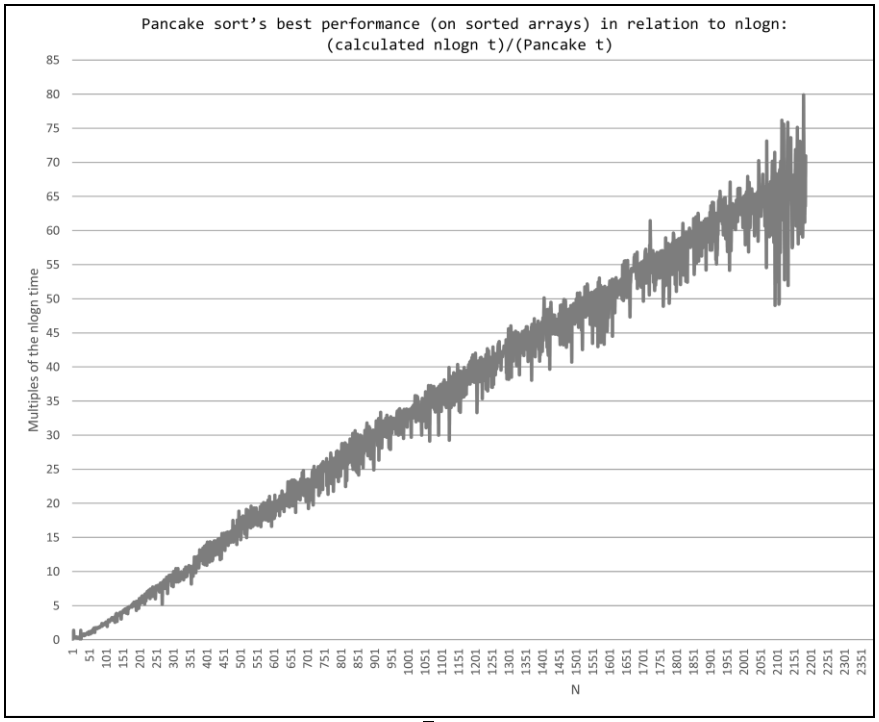

Figure 7 – When taking Bubble's measured $\overline{t}$ values on pre-sorted arrays as O(n) results, it is possible to calculate the $\overline{t}$ values for an $O(\log_2 N)$ scenario and use them as baseline for comparison with A's $\overline{t}$ values. This show that A's best performance has a steep linear difference from O($\log_2 N$).

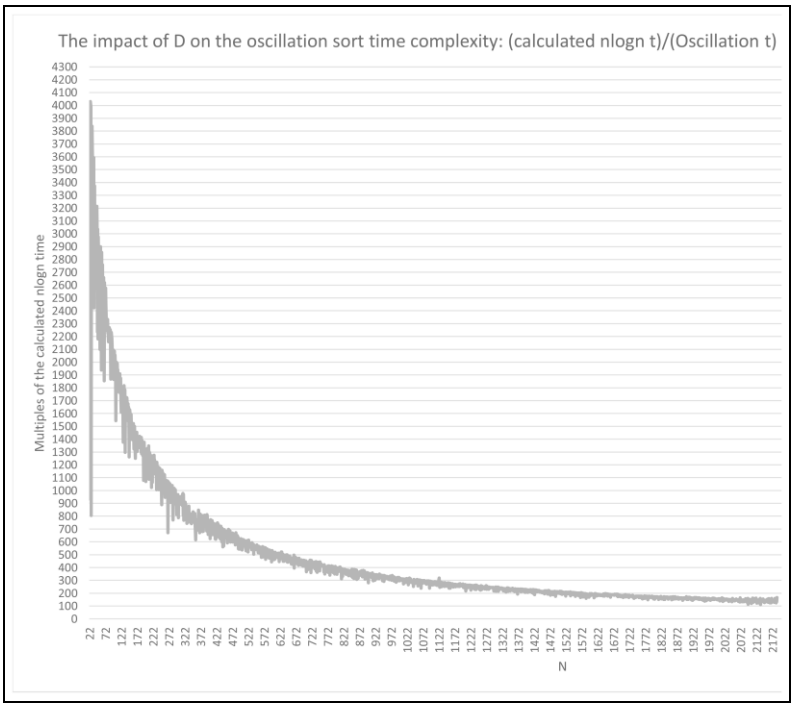

Figure 8 -Taking the same approach with B (as done with A on the previous figure) reveals how close B gets to $O(\mathrm{N}\log_2 N)$ as D decreases. Even when D reaches 0 at the last N on the figure, B still has a significant C=~150 multiplier.

1.3 Section E: Conclusion

The observed relative performance matches and further specifies the initial expectations:

- While both A and Bubble have $O(n^2)$, Bubble outperforms A on random arrays and with an increasing degree as the input array is more sorted. This tendency changes only when the input array is inversely sorted and Bubble performs at its worst.
- B's significant constant overhead and its performance dependence on D makes it the worst performing for relatively large D-s in terms of Δt. This is especially noticeable for small N-s (N<~1500), where its T=5 dominates the performance.

With the exception of Bubble's pre-sorted scenario of $O(N)$, none of the three algorithms were able to get considerably close to the best possible comparison sort performance of O(N logN).

Bubble's strength lies in pre-sorted (or almost pre-sorted) arrays, but suffers with reverse sorted arrays. A also performs best on pre-sorted (no flips are needed), but does not suffer with reverse sorted arrays. B is unaffected by data distribution and provides a constant performance between calculable N ranges. Ignoring Bubble's $O(N)$ scenario, B also provided the smallest memory access and comparison number by far. However, this advantage is cannot be turned into a positive impact on $\bar{t}$ when oscillation is used as an internal (in-memory) sort. Using slow external devices (e.g. tapes) would promote these features.

2 Appendix

2.1 Part 1: Source codes

2.1.1 pancakeSort.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:        Pancake Sort Performance Recorder in MATLAB
% Author:       SID: 1402187
% Rev. Date:    30 Apr 2016
% Original source:
http://rosettacode.org/wiki/Sorting_algorithms/Pancake_sort#MATLAB_.2F_Octav
e
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [V numComparisons numAccesses] = pancakeSort(V)

    numComparisons = 0;
    numAccesses = 0;

    for i = (numel(V):-1:2)


        numAccesses = numAccesses + 2;
        maxElem = V(i);
        maxIndex = i;

        %Find the max element in the current subset of the list
        for j = (i:-1:1)

            numComparisons = numComparisons +1;
            numAccesses = numAccesses +2;
            if V(j) >= maxElem
                numAccesses = numAccesses + 2;
                maxElem = V(j);
                maxIndex = j;
            end
        end

        %If the element is already in the correct position don't flip
        if i ~= maxIndex

            %First flip flips the max element in the stack to the top
            for a = 1:maxIndex
                numAccesses = numAccesses +4;
                swap = V(a);
                V(a) = V(maxIndex);
                V(maxIndex) = swap;
            end
```

```
            %Second flip flips the max element into the correct position in
            %the stack
            k = i;
            b = 1;
            while b<k
                numAccesses = numAccesses +4;
                swapb = V(b);
                V(b) = V(k);
                V(k) = swapb;
                k = k-1;
                b = b + 1;
            end

        end %end if
    end %for
end %pancakeSort
```

2.1.2 oscillatingSort.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:        Oscillating (Merge) Sort Performance Recorder in MATLAB
% Author:       SID: 1402187
% Rev. Date:    30 Apr 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [V numComparisons numAccesses] = oscillatingSort(V)

  numComparisons    = 0;
  numAccesses     = 0;

  %V will be filled up with this to have (N-1)^X values. Therefore can be
  %merged fully. These will be removed at the end.
  MAGICNUMBER =  intmin('int16');
  hasMagicUsed = false;

  SAFETOSTACK = 7; %Limits to (N-1)^7 maximum  sequence length.
                   %This implementation unable to guarantee consistency of
                   %sequence stacks above (N-1)^7 sequence lengths.
                   %(consistency: stacked sequences have descending length)

  inputLength = numel(V);
  N = 0;
  SetNumberOfTapesAndMagicNumbers; %This will set N (number of tapes). Min
4.

  SSSPLength = ceil(log(inputLength)/log(N))+1; %Top level sequence's length
  SSSP = NaN(N,SSSPLength);    %Sorted Sequence's Stored Power ("pyramid")
                               %Can be used to visually represent tape
                               %stacks.

  headPos = ones(N,1);  %Set each tape's head to the start position
  tapeMark = ones(N,1); %Guaranteed sorted order from beginning until
tapemark
  i = 0;                %The head position on the input tape (V)

  Tapes = NaN(N, inputLength); %Simulated merger tapes.
  MMT = 0;  %Main Merger Tape: where the next merging will be placed
  NTTW = 0; %Next Tape To Write:the next input read will be placed here


  barchartnumber = 0; %For plotting only.

  %MAIN LOOP%
  while i < inputLength
      SelectNextMainMergerTape_BeforePhase1;
```

```
        phase1InternalSort;
        phase2Merge;


        %for debug:
        %MYDEBUG = sum(~isnan(Tapes),2);
        %printCurrentStatus; %PLOTTING

        %If phase 3 merge(s) needed, calls them.
        numberP3 = GetRequiredPhase3Numer(i);
        iter = 1;
        while (numberP3 >= iter)

            if(iter == numberP3)
                [areEqualSequencesOnTop, extraP3Length] =
CheckEqualSequencesOnTop;
                if(areEqualSequencesOnTop == false)
                    break;
                end;
            end

                SelectNextMainMergerTape_ForPhase3(iter + 1);
                phase3Merge(power(N-1,iter+1));

                %for debug:
                %MYDEBUG = sum(~isnan(Tapes),2);
                %printCurrentStatus; %PLOTTING

            iter = iter + 1;
        end %while


        %Check if additional phase 3s become available
        [areEqualSequencesOnTop, extraP3Length] = CheckEqualSequencesOnTop;
        while (areEqualSequencesOnTop == true)
            SelectNextMainMergerTape_ForPhase3(extraP3Length +1);
            phase3Merge(power(N-1,extraP3Length + 1));

            %for debug:
            %MYDEBUG = sum(~isnan(Tapes),2);
            %printCurrentStatus; %PLOTTING

            [areEqualSequencesOnTop, extraP3Length] =
CheckEqualSequencesOnTop;
        end %while


        if(numberP3>0)
            TryToDefragment;
            %for debug:
            %MYDEBUG = sum(~isnan(Tapes),2);
```

```
            %printCurrentStatus; %PLOTTING
        end;
    end %while
    %END OF MAIN LOOP%



    %Phase 3 merge: merges "numItems" number of elements
    %into descending or ascending order.
    function phase3Merge(numItems)

        counters(1:N) = (numItems/(N-1));
        merges = 1;
        shouldTapeMarkAtEnd = false;
        %Taping at the end if MMT is empty
        if(headPos(MMT) == 1)
            shouldTapeMarkAtEnd = true;
        end %if

        %Check the current sorting order of a non MMT tape
        isInAscendingOrder = true;
        isOrderDetermined = false;
        aSample = 1;

        while(~isOrderDetermined && aSample <= N)
            if(aSample ~= MMT)
                numAccesses = numAccesses + 2;
                numComparisons = numComparisons + 1;
                if(Tapes(aSample, headPos(aSample) - 1) >  Tapes(aSample,
headPos(aSample)))
                    isInAscendingOrder = false;
                    isOrderDetermined = true;
                else
                   %if the two consequent number were equal, compare with the
beginning (spin the tape a lot)
                    numAccesses = numAccesses + 2;
                    numComparisons = numComparisons + 1;
                    if (Tapes(aSample, headPos(aSample) - 1) ==
Tapes(aSample, headPos(aSample)))
                         numAccesses = numAccesses + 2;
                         numComparisons = numComparisons + 1;
                         if (Tapes(aSample, headPos(aSample) - ((numItems/(N-
1))-1) ) >  Tapes(aSample, headPos(aSample)))
                              isInAscendingOrder = false;
                              isOrderDetermined = true;
                         else
                             numAccesses = numAccesses + 2;
                             numComparisons = numComparisons + 1;
                             if(Tapes(aSample, headPos(aSample) -
((numItems/(N-1))-1) ) ==  Tapes(aSample, headPos(aSample)))
                                 isOrderDetermined = false; %continue looking.
```

```
                        else
                            isOrderDetermined = true;
                        end %if
                    end %if
                else
                    isOrderDetermined = true;
                end %if
            end %if
        end %if
        aSample = aSample + 1;
    end %while
    %End of checking the sort order.


    while (merges <= numItems)

        defaultMaxFound = false;
        m = 1;
        while (~defaultMaxFound)
            if((m ~= MMT) && (counters(m) ~= 0))
                defaultMaxFound = true;
            else
                m = m + 1;
            end %if
        end %while

        numAccesses = numAccesses + 1;
        minmax = Tapes(m, headPos(m)); %Becomes default min value
        minmaxTapeIndex = m; %Becomes default min index
        m = m + 1; %skipping self-comparing

        %Maximum selection from "top on the stack" numbers
          while (m <= N)

            if(m ~= MMT)
               if(counters(m) ~= 0)

                  if(isInAscendingOrder == true)
                      numAccesses = numAccesses + 2;
                      numComparisons = numComparisons + 1;
                      if(Tapes(m, headPos(m)) > minmax)
                          numAccesses = numAccesses + 2;
                          minmax = Tapes(m, headPos(m));
                          minmaxTapeIndex = m;
                      end  %if
                  else
                      numAccesses = numAccesses + 2;
                      numComparisons = numComparisons + 1;
                      if(Tapes(m, headPos(m)) < minmax)
                          numAccesses = numAccesses + 2;
                          minmax = Tapes(m, headPos(m));
```

```
                            minmaxTapeIndex = m;
                        end  %if
                    end
                end %if
            end %if
            m = m + 1;
          end %while

        %Append the maximum into the main merger tape
        numAccesses = numAccesses + 1;
        if(~isnan(Tapes(MMT, headPos(MMT))))
             headPos(MMT) = headPos(MMT) + 1;
        end %end if
        numAccesses = numAccesses + 2;
        Tapes(MMT, headPos(MMT)) = minmax;

        %Delete the selected number from the tape where it found
        numAccesses = numAccesses + 1;
        Tapes(minmaxTapeIndex, headPos(minmaxTapeIndex)) = NaN;
        if(headPos(minmaxTapeIndex) ~= 1)
            headPos(minmaxTapeIndex) = headPos(minmaxTapeIndex) - 1;
        end %if
        counters(minmaxTapeIndex) = counters(minmaxTapeIndex) - 1;

        %One more merged
        merges = merges + 1;

    end % while

    %Setting tapemark if needed
    if(shouldTapeMarkAtEnd == true)
        tapeMark(MMT) = headPos(MMT);
    end %if
    ClearUpTapeMarks; %clears up tapemarks if needed

    %Updating Shortest Sorted Sequence Powers
     myit = 1;
     while (myit<=SSSPLength && ~isnan(SSSP(MMT,myit)))
          myit = myit + 1;
     end
     SSSP(MMT, myit) = floor(log(numItems)/log(N-1));



    myiter = 1;
    while myiter <= N
        if(myiter ~= MMT)
            if(headPos(myiter) == 1)
                SSSP(myiter,1) = NaN;
                SSSP(myiter,2:SSSPLength) = NaN;
            else
```

```
                    latestE = 1;
                    while(latestE <= SSSPLength &&
~isnan(SSSP(myiter,latestE)))
                       latestE = latestE + 1;
                    end

                    if(latestE ~= 1)
                       latestE = latestE-1;
                    end

                    SSSP(myiter, latestE) = NaN;

                end %if
            end%if
            myiter = myiter + 1;
        end %while

    end %function phase3Merge


  %Phase 2 merge: merges the latest N-1 read values in ascending order.
  function phase2Merge

          merges = 1;
          while merges <= (N-1)


              %Select default min value (non-NaN)
              defaultMinFound = false;
              m = 1;
              while ~defaultMinFound
                  numAccesses = numAccesses + 1;
                  numComparisons = numComparisons + 1;
                  if((m ~= MMT) && ~isnan(Tapes(m, headPos(m))))
                      defaultMinFound = true;
                  else
                     m = m + 1;
                  end%if
              end %while
              numAccesses = numAccesses + 1;
              min = Tapes(m, headPos(m)); %Becomes default min value
              minTapeIndex = m; %Becomes default min index
              m = m + 1; %skipping self-comparing


              %Minimum selection from the latest reads
                  while m <= N
                      if(m ~= MMT)
                          numAccesses = numAccesses + 1;
                          numComparisons = numComparisons + 1;
                          if(~isnan(Tapes(m, headPos(m))))
```

```
                                numAccesses = numAccesses + 2;
                                numComparisons = numComparisons + 1;
                                if(Tapes(m, headPos(m)) < min)
                                    numAccesses = numAccesses + 2;
                                    min = Tapes(m, headPos(m));
                                    minTapeIndex = m;
                                end  %if
                            end %if
                        end %if
                        m = m + 1;
                    end %while



                %Append the minimum into the main merger tape
                numAccesses = numAccesses + 1;
                numComparisons = numComparisons + 1;
                if(~isnan(Tapes(MMT, headPos(MMT))))
                  headPos(MMT) = headPos(MMT) + 1;
                end %end if
                numAccesses = numAccesses + 2;
                Tapes(MMT, headPos(MMT)) = min;


                %Delete the stored value from the tape where minimum found
                numAccesses = numAccesses + 1;
                Tapes(minTapeIndex, headPos(minTapeIndex)) = NaN;

                %One minimum found and merged.
                merges = merges + 1;


            end %while

            myit = 1;
            while (myit<=SSSPLength && ~isnan(SSSP(MMT,myit)))
               myit = myit + 1;
            end
            SSSP(MMT, myit) = 1;

            rewindNoneMainMergerWithOneStep;
    end %function

    %Multiples of pow(n-1,X) in "INPUTS" require X-1 "phase 3" merges
    function numP3 = GetRequiredPhase3Numer(INPUTS)
       isPh3Needed = (mod(INPUTS, power(N-1, 2)) == 0);

       if(isPh3Needed)
         numP3 = 2;
         pwrs = 2;
         while (power(N-1, pwrs) <= INPUTS)
```

```
            fraction = mod(INPUTS / power(N-1, pwrs), 1);
            if(fraction == 0)
                numP3 = pwrs;
            end%
            pwrs = pwrs + 1;
        end %while
        numP3 = numP3 - 1;

    else
        numP3 = 0;
    end %if
end %function

%Moves back the non Main Merger tapes' head with one.
function rewindNoneMainMergerWithOneStep
    tp = 1;
    while(tp <= N)
        if(tp ~= MMT && headPos(tp) ~= 1)
            headPos(tp) = headPos(tp) - 1;
        end
        tp = tp + 1;
    end % while
end %function

%Selects the next non main merger tape
 function FindNextTapeToWrite
     NTTW = mod(NTTW, N) + 1;
     if(NTTW == MMT)
         NTTW = mod(NTTW, N) + 1;
     end %if
 end %function FindNextTapeToWrite


 %Phase 1: Read in N-1 data from the input and writes to the tapes
 function phase1InternalSort
     j = 1;
     while j <= (N-1)
       i = i + 1; %Move input reader head

       FindNextTapeToWrite;

       %prevent overwriting if there is already a number at the head
       numAccesses = numAccesses + 1;
       numComparisons = numComparisons + 1;
       if(~isnan(Tapes(NTTW, headPos(NTTW))))
           headPos(NTTW) = headPos(NTTW) + 1;
       end %end if

       %Read from input, place to nextTapeToWrite
       numAccesses = numAccesses + 2;
       Tapes(NTTW, headPos(NTTW)) = V(i);
```

```
        j = j + 1;
      end %while
    end % function phase1InternalSort


    %Selects a merger tape suitable for an (N-1)^X element merging
    function SelectNextMainMergerTape_ForPhase3 (SequenceSizePower)

       %Find a place where SSSP is SequenceSizePower + 1.
       %If can't find, get one where <1
        isSuitableFound = false;
        secondaryOptionIndex = 0;
        tertiaryOptionIndex = 0;
        tertiaryOptionValue = NaN;
        iterator = 1;
        while (~isSuitableFound && iterator <= N)

            %Find the top on the stack element
            Jiterator = 1;
            while(Jiterator<=SSSPLength && ~isnan(SSSP(iterator,Jiterator)))
                Jiterator = Jiterator + 1;
            end%while

            if(Jiterator == 1)
                secondaryOptionIndex = iterator;
                iterator = iterator + 1;
                continue;
            end;

            %Check if this element is suitable
            if(Jiterator<=SSSPLength)
                Jiterator = Jiterator - 1;

                if(isnan(tertiaryOptionValue))
                   tertiaryOptionValue =  SSSP(iterator,Jiterator);
                   tertiaryOptionIndex = iterator;
                end

                if(SSSP(iterator,Jiterator) == (SequenceSizePower + 1) &&
iterator <= N)
                    isSuitableFound = true;
                    MMT = iterator;
                    break;
                end %if


                if(SSSP(iterator,Jiterator) > tertiaryOptionValue)
                    tertiaryOptionIndex = iterator;
                end %if
```

```
            end%if

            iterator = iterator + 1;
        end%while


        if(isSuitableFound == false)
            if(secondaryOptionIndex ~= 0)
                MMT = secondaryOptionIndex;
            else
                %No perfect matching space found.
                %Placing it on top of the longest sorted sequence (on top)
                MMT = tertiaryOptionIndex;
            end%if
        end



    end %SelectNextMainMergerTape_ForPhase3


    %Selects the merger tape for the upcoming (n-1) number to be read.
    function SelectNextMainMergerTape_BeforePhase1

        %At the start, select the first tape as MMT
        if(i == 0)
            MMT = 1;
            return;
        end;

        %Find a place where SSSP is 2. If can't find, get one where <1
        isSuitableFound = false;
        secondaryOptionIndex = 0;
        tertiaryOptionIndex = 0;
        tertiaryOptionValue = NaN;
        iterator = 1;
        while (~isSuitableFound && iterator <= N)

            %Find the top on the stack element
            Jiterator = 1;
            while(Jiterator<=SSSPLength && ~isnan(SSSP(iterator,Jiterator)))
                Jiterator = Jiterator + 1;
            end%while

            if(Jiterator == 1)
                secondaryOptionIndex = iterator;
                iterator = iterator + 1;
                continue;
            end;

            %Check if this element is suitable
```

```
            if(Jiterator<=SSSPLength)
                Jiterator = Jiterator - 1;

                if(isnan(tertiaryOptionValue))
                   tertiaryOptionValue =  SSSP(iterator,Jiterator);
                   tertiaryOptionIndex = iterator;
                end

                if(SSSP(iterator,Jiterator) == 2 && iterator <= N)
                    isSuitableFound = true;
                    MMT = iterator;
                    break;
                end %if

                if(SSSP(iterator,Jiterator) > tertiaryOptionValue)
                    tertiaryOptionIndex = iterator;
                end %if
            end%if

            iterator = iterator + 1;
        end%while


        if(isSuitableFound == false)
            if(secondaryOptionIndex ~= 0)
                MMT = secondaryOptionIndex;
            else
                %No perfect matching space found.
                %Placing it on top of the longest sorted sequence (on top)
                MMT = tertiaryOptionIndex;
            end%if
        end

    end %function SelectNextMainMergerTape_BeforePhase1


    %Selects the first tape which has its head position at the tape mark
    %to be the MMT. Returns false if can't.
    function isMarkedTapeWithSpaceFound = SelectMarkedTapeWithSpace
        isMarkedTapeWithSpaceFound = false;

        iterator = 1;
        while ((iterator <= N) && (isMarkedTapeWithSpaceFound == false))
            if(headPos(iterator) == tapeMark(iterator))
                isMarkedTapeWithSpaceFound = true;
                MMT = iterator;
            end %if
            iterator = iterator + 1;
        end %while
    end %function SelectEmptyTape
```

```
    %Selects the first empty tape to be the MMT. Returns false if can't.
    function isEmptyFound = SelectEmptyTape
        isEmptyFound = false;

        iterator = 1;
        while ((iterator <= N) && (isEmptyFound == false))
            if(headPos(iterator) == 1)
                isEmptyFound = true;
                MMT = iterator;
            end %if
            iterator = iterator + 1;
        end %while
    end %function SelectEmptyTape

    %Reutrns true if all non MMT tapes head position are on the sem level
    function result = isAllNonMMTOnSameLevel
        result = true;
        iterator = 1;
        Sample = 1;
        if(MMT == 1)
           Sample = 2;
        end

        while (iterator <= N)
            if(iterator ~= MMT)
                result = (result && (headPos(t) == Sample));
            end;
            iterator = iterator + 1;
        end%while
    end%

    %Selects the next tape in an oscillating manner (left to right)
    function SelectNextMainMergerTape_DefaultOscillating
        MMT = mod(MMT, N)+1;
    end %function SelectNextMainMergerTape_DefaultOscillating

    %Sets the tapemark value to 1 if a Tape's head at 1 too.
    function ClearUpTapeMarks
        iterator = 1;
        while (iterator <= N)
            if(headPos(iterator) == 1)
                tapeMark(iterator) = 1;
            end;
            iterator = iterator + 1;
        end %while
    end %function


    %Ensures that a single vector returned as V output without magic
numbers.
```

```
function FinishingUp

    hasDataCounter = 0;
    lastTapeWithData = 0;
    ab = 1;
    while(ab <= N)
        if(headPos(ab) ~= 1)
            hasDataCounter = hasDataCounter + 1;
            lastTapeWithData = ab;
        end
        ab = ab + 1;
    end%

    if(hasDataCounter == 1)
        if(hasMagicUsed == false)
            %Moving date from the tape to output
            numAccesses = numAccesses + (inputLength * 2);
            V = Tapes(lastTapeWithData,1:inputLength);
            %Check if flipping is needed to have ascending order
            numAccesses = numAccesses + 1;
            numComparisons = numComparisons + 1;
            if(V(1) > V(inputLength))
               numAccesses = numAccesses + (inputLength * 4); %inverting
               V = fliplr(V);
            end
        else
            numAccesses = numAccesses + (inputLength * 3);
            outputTape = Tapes(lastTapeWithData,1:inputLength);
            numComparisons = numComparisons + numel(outputTape);
            V = outputTape(outputTape~=MAGICNUMBER);
            numAccesses = numAccesses + numel(V);
            inputLength = numel(V);
            %Check if flipping is needed to have ascending order
            numComparisons = numComparisons + 1;
            if(V(1) > V(inputLength))
               numAccesses = numAccesses + (inputLength * 4); %inverting
               V = fliplr(V);
            end
        end
    else
        %This should not happen (always has to end with 1 column)
        V = Tapes;
    end;
end %function



%Checks if a tape starts with a sequence that has a length
%one power shorter than a top sequence on an another tape.
%If yes, calls the JoinTwoTapes function to merge them.
%
```

```
%NOTE on future improvements:
%This could be be further improved to chech not only the beginning of
%each tape, but looking at sub-sections of the tapes too. E.g.:
%TAPE1: (N-1)^4; (N-1)^2;
%TAPE2: (N-1)^3;
%TAPE3: (N-1)^1;
%TAPE4: EMPTY
%This would move ^1 to the top of ^2, but would miss the opportunity
%to move ^2 to ^3, thus might resulting unsolvable fregmentation
%down the road...
function TryToDefragment

    iterator = 1;
    while(iterator <= N)
        %If SSPN is empty: skip
        if(isnan(SSSP(iterator,1)))
            iterator = iterator + 1;
            continue;
        end;

        %Get the size of the last ordered block
        Jiterator = 1;
        while(Jiterator<=SSSPLength && ~isnan(SSSP(iterator,Jiterator)))
            Jiterator = Jiterator + 1;
        end%while
        Jiterator = Jiterator - 1;

        %If it is the smallest possible block: skip
        if(SSSP(iterator,Jiterator) == 1)
          iterator = iterator + 1;
          continue;
        end;

        %Find a tape that has a starting block 1 size bigger
        Miterator = 1;
        isTargetFound = false;
        while(Miterator <= N && ~isTargetFound)
            if(Miterator == iterator)
               Miterator = Miterator + 1;
               continue;
            end%if

            if(~isnan(SSSP(Miterator,1)))
                if(SSSP(Miterator,1) + 1 == SSSP(iterator,Jiterator))
                   isTargetFound = true;
                   break;
                end  %if
            end %if
            Miterator = Miterator + 1;
        end %if
```

```
            %If suitable target found, then defragment
            if(isTargetFound == true)
                JoinTwoTapes(iterator, Miterator);
                continue;
            end


            iterator = iterator + 1;
        end %while
    end


    %Joins two tapes by stacking one's content on top of the other.
    function JoinTwoTapes(staying, moving)

        %Copy tape values
        copyIter = 1;
        while(copyIter <= headPos(moving))
            headPos(staying) = headPos(staying) + 1;
            numAccesses = numAccesses + 3;
            Tapes(staying, headPos(staying)) = Tapes(moving, copyIter);
            Tapes(moving, copyIter) = NaN;
            copyIter = copyIter + 1;
        end%while

        %Set heads and tapes
        headPos(moving) = 1;
        tapeMark(moving) = 1;

        %Join SSSP values
        ssspStayingIter = 1;
        while(~isnan(SSSP(staying, ssspStayingIter)))

           ssspStayingIter = ssspStayingIter + 1;
        end

        ssspMovingIter = 1;

        while(~isnan(SSSP(moving, ssspMovingIter)))
            SSSP(staying, ssspStayingIter) = SSSP(moving, ssspMovingIter);
            SSSP(moving, ssspMovingIter) = NaN;
            ssspMovingIter = ssspMovingIter + 1;
            ssspStayingIter = ssspStayingIter + 1;
        end%while
    end%end function



    %Check if the (N-1) sequences on the top level have equal length.
    %If yes, returns the length of those sequences.
    function [hasEq, EqLength] = CheckEqualSequencesOnTop
```

```
        hasEq = false;
        EqLength = 0;

        %Get the length of the last sequence on each column
        lastSeqLengths = NaN(N,1);
        myi = 1;
        while(myi <= N )
            myx = 1;
            while(myx <= SSSPLength && ~isnan(SSSP(myi,myx)))
                lastSeqLengths(myi) = SSSP(myi,myx);
                myx = myx + 1;
            end
            myi = myi + 1;
        end%while

        %Check if the top sequences on all columns have the same length
        %with the exception of one column, If true, then merging is
possible.
        nancounter = 0;
        FL1counter = 1;
        foundLength2 = NaN;
        FL2counter = 0;
        foundLength1 = 0;
        myz = 1;

        if(~isnan(lastSeqLengths(1)))
            foundLength1 = lastSeqLengths(1);
            myz = 2;
        else
            nancounter = nancounter + 1;
            if(~isnan(lastSeqLengths(2)))
                foundLength1 = lastSeqLengths(2);
                myz = 3;
            else
                %There won't be an opportunity anyway now.
                nancounter = nancounter + 1;
            end
        end

        wasSearchSuccessfull = true;


        while(myz <= N && nancounter<2)
            if(nancounter == 2)
                %Two empty columns found. Cant have merging opportunity.
                wasSearchSuccessfull = false;
                break;
            end;

            if(isnan(lastSeqLengths(myz)))
               nancounter = nancounter + 1;
```

```
                myz = myz + 1;
                continue;
            end

            if(lastSeqLengths(myz) ~= foundLength1)
                if(isnan(foundLength2))
                    foundLength2 = lastSeqLengths(myz);
                    FL2counter = 1;
                else
                    if(lastSeqLengths(myz) ~= foundLength2)
                        %This is a third type of length. Cant have merging
                        %opportunity.
                        wasSearchSuccessfull = false;
                        break;
                    else
                       FL2counter = FL2counter + 1;
                    end
                end
            else
                 FL1counter = FL1counter + 1;
            end
            myz = myz + 1;
        end;%while

        if(nancounter == 1)
           if(~isnan(foundLength2))
               wasSearchSuccessfull = false;
           end
        end

        if(FL1counter > 1 && FL2counter > 1)
            wasSearchSuccessfull = false;
        end


        %Check if opportunity found
        if((myz>=N) && (nancounter < 2) && wasSearchSuccessfull == true)
            hasEq = true;
            if(FL1counter > FL2counter)
                EqLength = foundLength1;
            else
                EqLength = foundLength2;
            end
        end
    end


    %Fills up the V to have (n-1)^X numbers
    function SetNumberOfTapesAndMagicNumbers
        N = 4;
        while (inputLength > power(N-1,SAFETOSTACK))
```

```
            N = N + 1;
        end

        %If it is not exactly the same as the power number
        %then fill with magic numbers (which will be removed later)
        if(inputLength ~= power((N-1),SAFETOSTACK))

            desiredLength = power((N-1),SAFETOSTACK);
            differ = desiredLength - inputLength;
            magics(1:differ) = MAGICNUMBER;
            numAccesses = numAccesses + differ;
            V = [V magics];
            hasMagicUsed = true;
            inputLength = desiredLength;
        end
    end

    %Prints out the current state of the pyramid to a file.
    %Can be used to create an animation by concatenating the images.
    function printCurrentStatus
%         barchartnumber = barchartnumber + 1;
%         filename = strcat(num2str(barchartnumber),'_oscil.jpg');
%         fig = figure;
%         set(fig, 'Visible', 'off');
%         elementscount = (N-1).^SSSP;
%         bar(elementscount, 'stacked');
%         title(strcat('Oscillating (merge) sort        (i = ', num2str(i),
'; N = ', num2str(N),'; inputLength = ', num2str(inputLength), ';)'));
%         ylim([0 inputLength]);
%         xlabel('Merger tapes','FontSize', 12);
%         ylabel('Length of stacked sorted sequences','FontSize', 12);
%         saveas(fig, filename, 'png');
    end

     FinishingUp;
end % main function
```

2.1.3 testHarness.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:            Sort Algorithm Performance Test
% Author:           SID: 1402184
% Original author:  Ian van der Linde
% Rev. Date:        03 May 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;
close all;
clc;

maxArrayLength                = 100;
numRepeats                    = 10;

for currentArrayLength = 2:maxArrayLength

    %Bubble sort values
    bubbleSort_c_acc(1:numRepeats)       = 0;
    bubbleSort_m_acc(1:numRepeats)       = 0;
    bubbleSort_t_acc(1:numRepeats)       = 0;

    %Pancake sort values

    pencakeSort_c_acc(1:numRepeats)       = 0;
    pencakeSort_m_acc(1:numRepeats)       = 0;
    pencakeSort_t_acc(1:numRepeats)       = 0;

    %Oscillation sort values
    oscillationSort_c_acc(1:numRepeats)       = 0;
    oscillationSort_m_acc(1:numRepeats)       = 0;
    oscillationSort_t_acc(1:numRepeats)       = 0;


    for currentRepeat = 1:numRepeats

        %Using the same array for all the three algorithms
        testArray             = randi(30000,1,currentArrayLength);

        %Execute bubble sorting
        [v, c, m]             = bubbleSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = bubbleSort(testArray);
        bubbleSort_t_acc(currentRepeat)  = toc;
        bubbleSort_c_acc(currentRepeat)  = c;
        bubbleSort_m_acc(currentRepeat)  = m;

        %Execute pancake sorting
        [v, c, m]             = pancakeSort(testArray); % To Prime Cache
```

```
Memory
        tic;
        [v, c, m]             = pancakeSort(testArray);
        pancakeSort_t_acc(currentRepeat)  = toc;
        pancakeSort_c_acc(currentRepeat)  = c;
        pancakeSort_m_acc(currentRepeat)  = m;

        %Execute oscillation sorting
        [v, c, m]             = oscillatingSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = oscillatingSort(testArray);
        oscillationSort_t_acc(currentRepeat)  = toc;
        oscillationSort_c_acc(currentRepeat)  = c;
        oscillationSort_m_acc(currentRepeat)  = m;

    end

    %Bubble sort result means and std devs
    bubbleSort_elapsedTime_mean(currentArrayLength) =
mean(bubbleSort_t_acc);
    bubbleSort_elapsedTime_SD(currentArrayLength)   = std(bubbleSort_t_acc);

    bubbleSort_comparisons_mean(currentArrayLength) =
mean(bubbleSort_c_acc);
    bubbleSort_comparisons_SD(currentArrayLength)   = std(bubbleSort_c_acc);

    bubbleSort_memAccess_mean(currentArrayLength)   =
mean(bubbleSort_m_acc);
    bubbleSort_memAccess_SD(currentArrayLength)     = std(bubbleSort_m_acc);

    %Pancake sort result means and std devs
    pancakeSort_elapsedTime_mean(currentArrayLength) =
mean(pancakeSort_t_acc);
    pancakeSort_elapsedTime_SD(currentArrayLength)   =
std(pancakeSort_t_acc);

    pancakeSort_comparisons_mean(currentArrayLength) =
mean(pancakeSort_c_acc);
    pancakeSort_comparisons_SD(currentArrayLength)   =
std(pancakeSort_c_acc);

    pancakeSort_memAccess_mean(currentArrayLength)   =
mean(pancakeSort_m_acc);
    pancakeSort_memAccess_SD(currentArrayLength)     =
std(pancakeSort_m_acc);

    %Oscillation sort result means and std devs
    oscillationSort_elapsedTime_mean(currentArrayLength) =
mean(oscillationSort_t_acc);
    oscillationSort_elapsedTime_SD(currentArrayLength)   =
```

```
std(oscillationSort_t_acc);

    oscillationSort_comparisons_mean(currentArrayLength) =
mean(oscillationSort_c_acc);
    oscillationSort_comparisons_SD(currentArrayLength)   =
std(oscillationSort_c_acc);

    oscillationSort_memAccess_mean(currentArrayLength)   =
mean(oscillationSort_m_acc);
    oscillationSort_memAccess_SD(currentArrayLength)     =
std(oscillationSort_m_acc);



end

save('result.mat');
```

2.1.4 preSortedArrayTestHarness.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:            Pre-Sorted Array - Algorithm Performance Test
% Author:           SID: 1402184
% Original author:  Ian van der Linde
% Rev. Date:        30 Apr 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;
close all;
clc;

maxArrayLength                = 3000;
numRepeats                    = 10;


for currentArrayLength = 2:maxArrayLength

    %Bubble sort values
    bubbleSort_c_acc(1:numRepeats)       = 0;
    bubbleSort_m_acc(1:numRepeats)       = 0;
    bubbleSort_t_acc(1:numRepeats)       = 0;

    %Pancake sort values

    pencakeSort_c_acc(1:numRepeats)       = 0;
    pencakeSort_m_acc(1:numRepeats)       = 0;
    pencakeSort_t_acc(1:numRepeats)       = 0;

    %Oscillation sort values
    oscillationSort_c_acc(1:numRepeats)       = 0;
    oscillationSort_m_acc(1:numRepeats)       = 0;
    oscillationSort_t_acc(1:numRepeats)       = 0;


    for currentRepeat = 1:numRepeats

        %Using the same array for all the three algorithms
        startpos              = randi(10000,1,1)+1;
        testArray             = (startpos:(startpos+currentArrayLength)-1);

        %Execute bubble sorting
        [v, c, m]             = bubbleSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = bubbleSort(testArray);
        bubbleSort_t_acc(currentRepeat)  = toc;
        bubbleSort_c_acc(currentRepeat)  = c;
        bubbleSort_m_acc(currentRepeat)  = m;

        %Execute pancake sorting
```

```
        [v, c, m]             = pancakeSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = pancakeSort(testArray);
        pancakeSort_t_acc(currentRepeat)  = toc;
        pancakeSort_c_acc(currentRepeat)  = c;
        pancakeSort_m_acc(currentRepeat)  = m;

        %Execute oscillation sorting
        [v, c, m]             = oscillatingSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = oscillatingSort(testArray);
        oscillationSort_t_acc(currentRepeat)  = toc;
        oscillationSort_c_acc(currentRepeat)  = c;
        oscillationSort_m_acc(currentRepeat)  = m;

    end

    %Bubble sort result means and std devs
    bubbleSort_elapsedTime_mean(currentArrayLength) =
mean(bubbleSort_t_acc);
    bubbleSort_elapsedTime_SD(currentArrayLength)   = std(bubbleSort_t_acc);

    bubbleSort_comparisons_mean(currentArrayLength) =
mean(bubbleSort_c_acc);
    bubbleSort_comparisons_SD(currentArrayLength)   = std(bubbleSort_c_acc);

    bubbleSort_memAccess_mean(currentArrayLength)   =
mean(bubbleSort_m_acc);
    bubbleSort_memAccess_SD(currentArrayLength)     = std(bubbleSort_m_acc);

    %Pancake sort result means and std devs
    pancakeSort_elapsedTime_mean(currentArrayLength) =
mean(pancakeSort_t_acc);
    pancakeSort_elapsedTime_SD(currentArrayLength)   =
std(pancakeSort_t_acc);

    pancakeSort_comparisons_mean(currentArrayLength) =
mean(pancakeSort_c_acc);
    pancakeSort_comparisons_SD(currentArrayLength)   =
std(pancakeSort_c_acc);

    pancakeSort_memAccess_mean(currentArrayLength)   =
mean(pancakeSort_m_acc);
    pancakeSort_memAccess_SD(currentArrayLength)     =
std(pancakeSort_m_acc);

    %Oscillation sort result means and std devs
    oscillationSort_elapsedTime_mean(currentArrayLength) =
mean(oscillationSort_t_acc);
```

```
    oscillationSort_elapsedTime_SD(currentArrayLength)   =
std(oscillationSort_t_acc);

    oscillationSort_comparisons_mean(currentArrayLength) =
mean(oscillationSort_c_acc);
    oscillationSort_comparisons_SD(currentArrayLength)   =
std(oscillationSort_c_acc);

    oscillationSort_memAccess_mean(currentArrayLength)   =
mean(oscillationSort_m_acc);
    oscillationSort_memAccess_SD(currentArrayLength)     =
std(oscillationSort_m_acc);


    disp(strcat('PSATH Completed: ', num2str((currentArrayLength /
maxArrayLength)*100),' %'))

end

save('presortedResult.mat');
```

2.1.5 nearlySortedArrayTestHarness10.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:            Nearly Sorted (10% salt) Array - Algorithm Performance
Test
% Author:           SID: 1402184
% Original author:  Ian van der Linde
% Rev. Date:        03 May 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;
close all;
clc;

maxArrayLength                = 3000;
numRepeats                    = 10;

saltationFrequency           = 10; %every n-th element sets to be random

for currentArrayLength = 2:maxArrayLength

    %Bubble sort values
    bubbleSort_c_acc(1:numRepeats)       = 0;
    bubbleSort_m_acc(1:numRepeats)       = 0;
    bubbleSort_t_acc(1:numRepeats)       = 0;

    %Pancake sort values

    pencakeSort_c_acc(1:numRepeats)       = 0;
    pencakeSort_m_acc(1:numRepeats)       = 0;
    pencakeSort_t_acc(1:numRepeats)       = 0;

    %Oscillation sort values
    oscillationSort_c_acc(1:numRepeats)       = 0;
    oscillationSort_m_acc(1:numRepeats)       = 0;
    oscillationSort_t_acc(1:numRepeats)       = 0;


    for currentRepeat = 1:numRepeats

        %Using the same array for all the three algorithms
        startpos              = randi(10000,1,1)+1;
        salterArray           = randi(10000,1,currentArrayLength);
        preorderedArray       = (startpos:(startpos+currentArrayLength)-1);
        testArray             = preorderedArray;

        %Salting the test array
        mysalter = saltationFrequency;
        while mysalter < length(testArray)
            testArray(mysalter) = salterArray(mysalter);
            mysalter = mysalter + saltationFrequency;
```

```
        end


        %Execute bubble sorting
        [v, c, m]             = bubbleSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = bubbleSort(testArray);
        bubbleSort_t_acc(currentRepeat)  = toc;
        bubbleSort_c_acc(currentRepeat)  = c;
        bubbleSort_m_acc(currentRepeat)  = m;

        %Execute pancake sorting
        [v, c, m]             = pancakeSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = pancakeSort(testArray);
        pancakeSort_t_acc(currentRepeat)  = toc;
        pancakeSort_c_acc(currentRepeat)  = c;
        pancakeSort_m_acc(currentRepeat)  = m;

        %Execute oscillation sorting
        [v, c, m]             = oscillatingSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = oscillatingSort(testArray);
        oscillationSort_t_acc(currentRepeat)  = toc;
        oscillationSort_c_acc(currentRepeat)  = c;
        oscillationSort_m_acc(currentRepeat)  = m;

    end

    %Bubble sort result means and std devs
    bubbleSort_elapsedTime_mean(currentArrayLength) =
mean(bubbleSort_t_acc);
    bubbleSort_elapsedTime_SD(currentArrayLength)   = std(bubbleSort_t_acc);

    bubbleSort_comparisons_mean(currentArrayLength) =
mean(bubbleSort_c_acc);
    bubbleSort_comparisons_SD(currentArrayLength)   = std(bubbleSort_c_acc);

    bubbleSort_memAccess_mean(currentArrayLength)   =
mean(bubbleSort_m_acc);
    bubbleSort_memAccess_SD(currentArrayLength)     = std(bubbleSort_m_acc);

    %Pancake sort result means and std devs
    pancakeSort_elapsedTime_mean(currentArrayLength) =
mean(pancakeSort_t_acc);
    pancakeSort_elapsedTime_SD(currentArrayLength)   =
std(pancakeSort_t_acc);
```

```
    pancakeSort_comparisons_mean(currentArrayLength) =
mean(pancakeSort_c_acc);
    pancakeSort_comparisons_SD(currentArrayLength)   =
std(pancakeSort_c_acc);

    pancakeSort_memAccess_mean(currentArrayLength)   =
mean(pancakeSort_m_acc);
    pancakeSort_memAccess_SD(currentArrayLength)     =
std(pancakeSort_m_acc);

    %Oscillation sort result means and std devs
    oscillationSort_elapsedTime_mean(currentArrayLength) =
mean(oscillationSort_t_acc);
    oscillationSort_elapsedTime_SD(currentArrayLength)   =
std(oscillationSort_t_acc);

    oscillationSort_comparisons_mean(currentArrayLength) =
mean(oscillationSort_c_acc);
    oscillationSort_comparisons_SD(currentArrayLength)   =
std(oscillationSort_c_acc);

    oscillationSort_memAccess_mean(currentArrayLength)   =
mean(oscillationSort_m_acc);
    oscillationSort_memAccess_SD(currentArrayLength)     =
std(oscillationSort_m_acc);


    disp(strcat('NSATH10 Completed: ', num2str((currentArrayLength /
maxArrayLength)*100),' %'))

end

save('nearlysortedresult10.mat');
```

2.1.6 nearlySortedArrayTestHarness20.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:            Nearly Sorted (20% salt) Array - Algorithm Performance
Test
% Author:           SID: 1402184
% Original author:  Ian van der Linde
% Rev. Date:        03 May 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;
close all;
clc;

maxArrayLength                = 3000;
numRepeats                    = 10;

saltationFrequency           = 5; %every n-th element sets to be random

for currentArrayLength = 2:maxArrayLength

    %Bubble sort values
    bubbleSort_c_acc(1:numRepeats)       = 0;
    bubbleSort_m_acc(1:numRepeats)       = 0;
    bubbleSort_t_acc(1:numRepeats)       = 0;

    %Pancake sort values

    pencakeSort_c_acc(1:numRepeats)       = 0;
    pencakeSort_m_acc(1:numRepeats)       = 0;
    pencakeSort_t_acc(1:numRepeats)       = 0;

    %Oscillation sort values
    oscillationSort_c_acc(1:numRepeats)       = 0;
    oscillationSort_m_acc(1:numRepeats)       = 0;
    oscillationSort_t_acc(1:numRepeats)       = 0;


    for currentRepeat = 1:numRepeats

        %Using the same array for all the three algorithms
        startpos              = randi(10000,1,1)+1;
        salterArray           = randi(10000,1,currentArrayLength);
        preorderedArray       = (startpos:(startpos+currentArrayLength)-1);
        testArray             = preorderedArray;

        %Salting the test array
        mysalter = saltationFrequency;
        while mysalter < length(testArray)
            testArray(mysalter) = salterArray(mysalter);
            mysalter = mysalter + saltationFrequency;
```

```
        end


        %Execute bubble sorting
        [v, c, m]             = bubbleSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = bubbleSort(testArray);
        bubbleSort_t_acc(currentRepeat)  = toc;
        bubbleSort_c_acc(currentRepeat)  = c;
        bubbleSort_m_acc(currentRepeat)  = m;

        %Execute pancake sorting
        [v, c, m]             = pancakeSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = pancakeSort(testArray);
        pancakeSort_t_acc(currentRepeat)  = toc;
        pancakeSort_c_acc(currentRepeat)  = c;
        pancakeSort_m_acc(currentRepeat)  = m;

        %Execute oscillation sorting
        [v, c, m]             = oscillatingSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = oscillatingSort(testArray);
        oscillationSort_t_acc(currentRepeat)  = toc;
        oscillationSort_c_acc(currentRepeat)  = c;
        oscillationSort_m_acc(currentRepeat)  = m;

    end

    %Bubble sort result means and std devs
    bubbleSort_elapsedTime_mean(currentArrayLength) =
mean(bubbleSort_t_acc);
    bubbleSort_elapsedTime_SD(currentArrayLength)   = std(bubbleSort_t_acc);

    bubbleSort_comparisons_mean(currentArrayLength) =
mean(bubbleSort_c_acc);
    bubbleSort_comparisons_SD(currentArrayLength)   = std(bubbleSort_c_acc);

    bubbleSort_memAccess_mean(currentArrayLength)   =
mean(bubbleSort_m_acc);
    bubbleSort_memAccess_SD(currentArrayLength)     = std(bubbleSort_m_acc);

    %Pancake sort result means and std devs
    pancakeSort_elapsedTime_mean(currentArrayLength) =
mean(pancakeSort_t_acc);
    pancakeSort_elapsedTime_SD(currentArrayLength)   =
std(pancakeSort_t_acc);
```

```
    pancakeSort_comparisons_mean(currentArrayLength) =
mean(pancakeSort_c_acc);
    pancakeSort_comparisons_SD(currentArrayLength)   =
std(pancakeSort_c_acc);

    pancakeSort_memAccess_mean(currentArrayLength)   =
mean(pancakeSort_m_acc);
    pancakeSort_memAccess_SD(currentArrayLength)     =
std(pancakeSort_m_acc);

    %Oscillation sort result means and std devs
    oscillationSort_elapsedTime_mean(currentArrayLength) =
mean(oscillationSort_t_acc);
    oscillationSort_elapsedTime_SD(currentArrayLength)   =
std(oscillationSort_t_acc);

    oscillationSort_comparisons_mean(currentArrayLength) =
mean(oscillationSort_c_acc);
    oscillationSort_comparisons_SD(currentArrayLength)   =
std(oscillationSort_c_acc);

    oscillationSort_memAccess_mean(currentArrayLength)   =
mean(oscillationSort_m_acc);
    oscillationSort_memAccess_SD(currentArrayLength)     =
std(oscillationSort_m_acc);


    disp(strcat('NSATH20 Completed: ', num2str((currentArrayLength /
maxArrayLength)*100),' %'))

end

save('nearlysortedresult20.mat');
```

2.1.7 reverseSortedArrayTestHarness.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:            Reverse Sorted Array - Algorithm Performance Test
% Author:           SID: 1402184
% Original author:  Ian van der Linde
% Rev. Date:        03 May 2016
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;
close all;
clc;

maxArrayLength                = 3000;
numRepeats                    = 10;

for currentArrayLength = 2:maxArrayLength

    %Bubble sort values
    bubbleSort_c_acc(1:numRepeats)       = 0;
    bubbleSort_m_acc(1:numRepeats)       = 0;
    bubbleSort_t_acc(1:numRepeats)       = 0;

    %Pancake sort values

    pencakeSort_c_acc(1:numRepeats)       = 0;
    pencakeSort_m_acc(1:numRepeats)       = 0;
    pencakeSort_t_acc(1:numRepeats)       = 0;

    %Oscillation sort values
    oscillationSort_c_acc(1:numRepeats)       = 0;
    oscillationSort_m_acc(1:numRepeats)       = 0;
    oscillationSort_t_acc(1:numRepeats)       = 0;


    for currentRepeat = 1:numRepeats

        %Using the same array for all the three algorithms
        startpos              = randi(10000,1,1)+10000;
        testArray             = (startpos:-1:(startpos-
currentArrayLength)+1);

        %Execute bubble sorting
        [v, c, m]             = bubbleSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = bubbleSort(testArray);
        bubbleSort_t_acc(currentRepeat)  = toc;
        bubbleSort_c_acc(currentRepeat)  = c;
        bubbleSort_m_acc(currentRepeat)  = m;
```

```
        %Execute pancake sorting
        [v, c, m]             = pancakeSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = pancakeSort(testArray);
        pancakeSort_t_acc(currentRepeat)  = toc;
        pancakeSort_c_acc(currentRepeat)  = c;
        pancakeSort_m_acc(currentRepeat)  = m;

        %Execute oscillation sorting
        [v, c, m]             = oscillatingSort(testArray); % To Prime Cache
Memory
        tic;
        [v, c, m]             = oscillatingSort(testArray);
        oscillationSort_t_acc(currentRepeat)  = toc;
        oscillationSort_c_acc(currentRepeat)  = c;
        oscillationSort_m_acc(currentRepeat)  = m;

    end

    %Bubble sort result means and std devs
    bubbleSort_elapsedTime_mean(currentArrayLength) =
mean(bubbleSort_t_acc);
    bubbleSort_elapsedTime_SD(currentArrayLength)   = std(bubbleSort_t_acc);

    bubbleSort_comparisons_mean(currentArrayLength) =
mean(bubbleSort_c_acc);
    bubbleSort_comparisons_SD(currentArrayLength)   = std(bubbleSort_c_acc);

    bubbleSort_memAccess_mean(currentArrayLength)   =
mean(bubbleSort_m_acc);
    bubbleSort_memAccess_SD(currentArrayLength)     = std(bubbleSort_m_acc);

    %Pancake sort result means and std devs
    pancakeSort_elapsedTime_mean(currentArrayLength) =
mean(pancakeSort_t_acc);
    pancakeSort_elapsedTime_SD(currentArrayLength)   =
std(pancakeSort_t_acc);

    pancakeSort_comparisons_mean(currentArrayLength) =
mean(pancakeSort_c_acc);
    pancakeSort_comparisons_SD(currentArrayLength)   =
std(pancakeSort_c_acc);

    pancakeSort_memAccess_mean(currentArrayLength)   =
mean(pancakeSort_m_acc);
    pancakeSort_memAccess_SD(currentArrayLength)     =
std(pancakeSort_m_acc);

    %Oscillation sort result means and std devs
    oscillationSort_elapsedTime_mean(currentArrayLength) =
```

```
mean(oscillationSort_t_acc);
    oscillationSort_elapsedTime_SD(currentArrayLength)   =
std(oscillationSort_t_acc);

    oscillationSort_comparisons_mean(currentArrayLength) =
mean(oscillationSort_c_acc);
    oscillationSort_comparisons_SD(currentArrayLength)   =
std(oscillationSort_c_acc);

    oscillationSort_memAccess_mean(currentArrayLength)   =
mean(oscillationSort_m_acc);
    oscillationSort_memAccess_SD(currentArrayLength)     =
std(oscillationSort_m_acc);


    disp(strcat('RSATH Completed: ', num2str((currentArrayLength /
maxArrayLength)*100),' %'))
end

save('reversedResult.mat');
```

2.1.8 figureCreator.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title:            Figure creator for sorting test harnesses
% Author:           SID: 1402184
% Original author:  Ian van der Linde
% Rev. Date:        03 May 2016
%
% Load in the saved sorting test result workspaces to generate the plots.
% If printing needed, uncomment the last line and set the file name
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


presortedFig = figure;
presortedFig.PaperType = 'a4';
presortedFig.PaperOrientation = 'portrait';
errorbarGapSize = 200;

%%Comparisons
subplot(3,1,1);
%Bubble
p1 = plot(1:maxArrayLength, bubbleSort_comparisons_mean,'LineWidth', 3);hold
on;
errorbar(1:errorbarGapSize:maxArrayLength,
bubbleSort_comparisons_mean(1:errorbarGapSize:end),
bubbleSort_comparisons_SD(1:errorbarGapSize:end), 'ko');hold on;
%Pancake
p2 = plot(1:maxArrayLength, pancakeSort_comparisons_mean,'LineWidth',
3);hold on;
errorbar(1:errorbarGapSize:maxArrayLength,
pancakeSort_comparisons_mean(1:errorbarGapSize:end),
pancakeSort_comparisons_SD(1:errorbarGapSize:end), 'ko');
%Oscillation
p3 = plot(1:maxArrayLength, oscillationSort_comparisons_mean,'LineWidth',
3);hold on;
errorbar(1:errorbarGapSize:maxArrayLength,
oscillationSort_comparisons_mean(1:errorbarGapSize:end),
oscillationSort_comparisons_SD(1:errorbarGapSize:end), 'ko');
legend([p1,p2,p3],'Bubble','Pancake','Oscillating','Location','northwest');

xlabel('Array Length','FontSize', 10);
ylabel('Comparisons','FontSize', 11);
xlim([0 maxArrayLength]);

%ylim
bestForCompareYLim = (1:3);
bestForCompareYLim(1) = max(bubbleSort_comparisons_mean);
```

```
bestForCompareYLim(2) = max(pancakeSort_comparisons_mean);
bestForCompareYLim(3) = max(oscillationSort_comparisons_mean);
ylim([0 max(bestForCompareYLim)]);


%%Memory accesses
```

```
subplot(3,1,2);
%Bubble
p4 = plot(1:maxArrayLength, bubbleSort_memAccess_mean,'LineWidth',3);hold
on;
errorbar(1:errorbarGapSize:maxArrayLength,
bubbleSort_memAccess_mean(1:errorbarGapSize:end),
bubbleSort_memAccess_SD(1:errorbarGapSize:end), 'ko');
%Pancake
p5 = plot(1:maxArrayLength, pancakeSort_memAccess_mean,'LineWidth',3);hold
on;
errorbar(1:errorbarGapSize:maxArrayLength,
pancakeSort_memAccess_mean(1:errorbarGapSize:end),
pancakeSort_memAccess_SD(1:errorbarGapSize:end), 'ko');
%Oscillation
p6 = plot(1:maxArrayLength,
oscillationSort_memAccess_mean,'LineWidth',3);hold on;
errorbar(1:errorbarGapSize:maxArrayLength,
oscillationSort_memAccess_mean(1:errorbarGapSize:end),
oscillationSort_memAccess_SD(1:errorbarGapSize:end), 'ko');
legend([p4,p5,p6],'Bubble','Pancake','Oscillating','Location','northwest');

xlabel('Array Length','FontSize', 10);
ylabel('Memory Accesses','FontSize', 11);
xlim([0 maxArrayLength]);

%ylim
bestForAccessYLim = (1:3);
bestForAccessYLim(1) = max(bubbleSort_memAccess_mean);
bestForAccessYLim(2) = max(pancakeSort_memAccess_mean);
bestForAccessYLim(3) = max(oscillationSort_memAccess_mean);
ylim([0 max(bestForAccessYLim)]);


%%Time elapsed
```

```
subplot(3,1,3);
%Bubble
p7 = plot(1:maxArrayLength, bubbleSort_elapsedTime_mean, 'LineWidth',3);hold
on;
errorbar(1:errorbarGapSize:maxArrayLength,
bubbleSort_elapsedTime_mean(1:errorbarGapSize:end),
bubbleSort_elapsedTime_SD(1:errorbarGapSize:end), 'ko');
%Pancake
p8 = plot(1:maxArrayLength, pancakeSort_elapsedTime_mean,
'LineWidth',3);hold on;
```

```
errorbar(1:errorbarGapSize:maxArrayLength,
pancakeSort_elapsedTime_mean(1:errorbarGapSize:end),
pancakeSort_elapsedTime_SD(1:errorbarGapSize:end), 'ko');
%Oscillation
p9 = plot(1:maxArrayLength, oscillationSort_elapsedTime_mean,
'LineWidth',3);hold on;
errorbar(1:errorbarGapSize:maxArrayLength,
oscillationSort_elapsedTime_mean(1:errorbarGapSize:end),
oscillationSort_elapsedTime_SD(1:errorbarGapSize:end), 'ko');
legend([p7,p8,p9],'Bubble','Pancake','Oscillating','Location','northwest');

xlabel('Array Length','FontSize', 10);
ylabel('Elapsed Time (s)','FontSize', 11);
xlim([0 maxArrayLength]);
ylim([0 max(pancakeSort_elapsedTime_mean)]);

%ylim
bestForTimeYLim = (1:3);
bestForTimeYLim(1) = max(bubbleSort_elapsedTime_mean);
bestForTimeYLim(2) = max(pancakeSort_elapsedTime_mean);
bestForTimeYLim(3) = max(oscillationSort_elapsedTime_mean);
ylim([0 max(bestForTimeYLim)]);
```

```
%print -f1 -dpng -r1200 sortResults.png
```

3 References

Dweighter, H., 1975. Elementary Problem E2569. *American Mathematical Monthly,* 82(1), p. 1010.

Knuth, D. E., 1998. *The Art of Computer Programming.* 2nd ed. Stanford: Addison-Wesley.

Sobel, S., 1962. Oscillating Sort–A New Sort Merging Technique. *Journal of the ACM,* 9(3), pp. 372-374.