Assignment #2 Memory

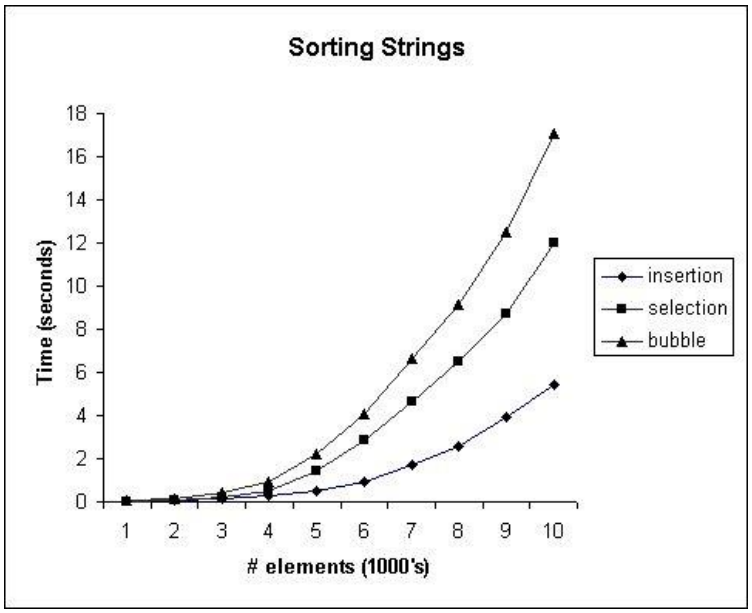
Introduction

Each problem in this assignment was based on problems in memory. At the beginning of this assignment I felt a sorting algorithm was necessary to make my solutions to these problems much more elegant, also using a sorting algorithm had the side benefit of outputting a fully sorted list for the user to use in the future. This side benefit can be seen especially useful in "Sets – Closure" and "Sets – Symmetric Difference", where the outputted sets are no longer randomised, but sorted mathematically.

During the course of this assignment, I research many possible sorting algorithms, I eventually decided upon using insertion sort. Due to the prominence of insertion sort throughout each stage of my code, I have dedicated an entire section of this report just to insertion sort, which I shall reference with suitable additional notes for each given problem.

Insertion Sort:

I decided to use insertion sort throughout this assignment for various reasons. I could have easily used bubble sort instead however I wanted to give myself a challenge by implementing a more efficient algorithm, albeit slightly more complex to write than bubble sort. In the case of the problems given for this assignment the efficiency benefits would realistically be minimal, as we are not sorting a large amount of data. However from the graph below we can see the benefits of using insertion sort over bubble sort and selection sort, if we were comparing even larger amounts of data.



Source of above graph: https://users.cs.duke.edu/~ola/bubble/bubble.html
Owen Astrachan, Computer Science Department, Duke University

Trinity College Dublin
The University of Dublin

School of Computer Science and Statistics
CS1021 - Introduction to Computing I
Conor Gildea - [redacted]
Assignment #2

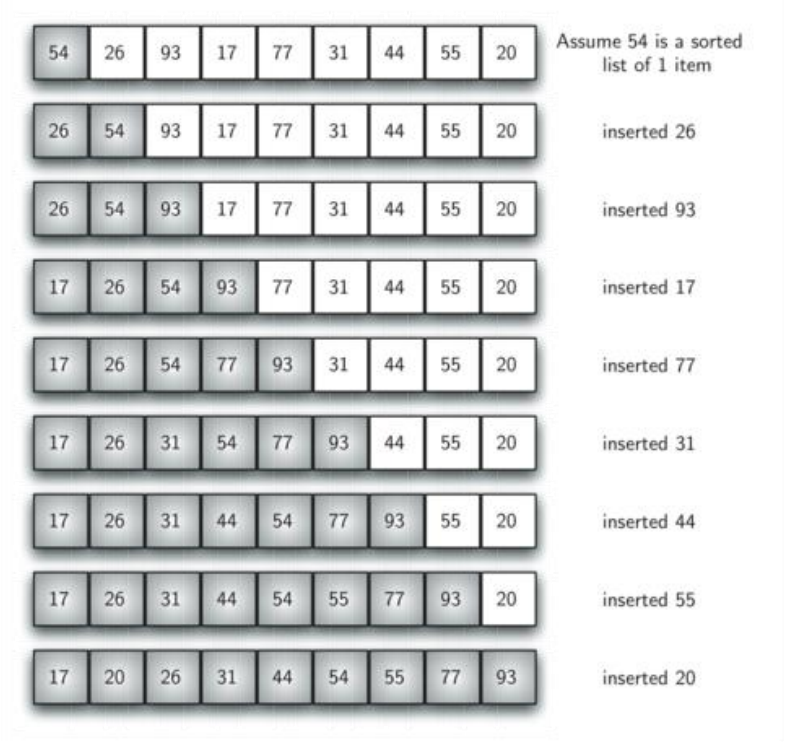
How does insertion sort work?

When are given an unsorted list we set the first element in the set to automatically be in the sorted section of the list, we then consider all other elements to be in the unsorted section. We compare the first element we collected, with the first element in the unsorted section. If the element from the unsorted section is smaller than the element we currently have we swap them, if not we move on to compare the next element in the unsorted list.

We now have two elements in our sorted list and we compare them with that next element gotten from the unsorted section. Yet again if that next element in the unsorted section happened to be smaller than the last element in the sorted section, we swap them.

Now this is what makes insertion sort special is the fact, if that next unsorted element was smaller than our highest element in our sorted section, we then compare it with our first element in our sorted section. Yet again, if it is smaller we swap them, if not the number is inserted into the correct place on the list between the first and second sorted numbers. The process is then repeated for all elements until all elements have been sorted. This is how insertion sort works.

Please consult the below image for a visual representation of how insertion sort works:



Source of the above image: https://interactivepython.org/runestone/static/pythonds/SortSearch/TheInsertionSort.html

It is interesting to note the final two lines of numbers in the above image. Here we can clearly see the number 20, moving from the unsorted section, in the second final line, to being inserted to become the second number in the series, in the final line. This yet again shows us how insertion sort works.

Insertion Sort Psuedocode

The below pseudocode was sourced from Harvard's CS50 Youtube video "Insertion Sort"
Source: https://www.youtube.com/watch?v=DFG-XuyPYUQ

For i = 1 to n-1

element = array[i]

j = i

while (j>0 and array[j-1]>element)

array[j] = array[j-1]

j=j-1

array[j] = element

The above pseudocode could be used in a higher level language however for the purposes of this assignment I had to break down the insertion sort algorithm, for use in a lower level language such as ARM assembly.

```
while(aChangingAddress!=finishedAddress)
{
    currentValue = Memory.word(aChangingAddress);
    testValue = Memory.word(testValueAddress);


    if(currentValue>testValue)
    {
        Memory.word(testValueAddress) = currentValue;
        Memory.word(aChangingAddress) = testValue;
        if(startingAddress!=testValueAddress)
    {
            changingAddressBeforeInsertion = aChangingAddress;
            insertionCheckingAddress = aChangingAddress - 4;

            while((startingAddress!=testValueAddress)&&(testValue<insertionCurrentValue)
            {
                insertionCurrentValue = Memory.word(insertionCheckingAddress) // This line in code is before the
                                                                              // second condition of the while
                                                                              // statement is met
                if(testValue<insertionCurrentValue)
                {
                    Memory.word(insertionCheckingAddress) = testValue;
                    Memory.word(aChangingAddress) = insertionCurrentValue;
                    insertionCheckingAddress -=4;
                    aChangingAddress -= 4;
                    testValue = Memory.word(aChangingAddress);

                }
            }
            aChangingAddress = changingAddressBeforeInsertion;
        }
    }
currentValueAddress += 4;
testValueAddress += 4;
currentValue = 0;
testValue = 0;
//Branch unconditionally to sorting
}
```

The above pseudocode is from my solution from Sets-Closure however an almost identical copy of this insertion sort algorithm can be found in my other two solutions in this assignment, the only differences would really be different address names and there is different address increments for the final problem "Anagrams" (All 4s replaced with 1s).

It is interesting to note the differences and similarities between the high level pseudocode and the lower level ARM assembly pseudocode, what stands out is the major similarities of both having two loops, one being a nested loop. One of the main differences would be my implementation of addresses instead of arrays.

My pseudocode works by first getting a currentValue and a testValue, the testValue is in the next address above the currentValue. If the testValue happens to be smaller, the testValue is then compared with all the previous numbers in the sorted section of the set. Once it is no longer smaller than the previous sorted number, or it is now the lowest number in the set, it inserts the number into that position. I use the word insert however realistically, the program is just constantly swapping or not swapping two numbers beside each other, until it reaches a number that is higher than testValue.

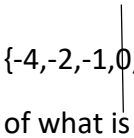
After testValue has been inserted, the algoritm moves up to the next address and loads testValue to be the previous testValue address plus the address increment for the program and also loads currentValue to be the previous currentValue address plus the address increment for the program. The program then loops around again and continues until all elements have been sorted and it meets the end of the set.

Trinity College Dublin
The University of Dublin

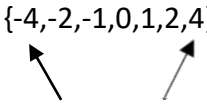
School of Computer Science and Statistics
CS1021 - Introduction to Computing I
Conor Gildea - [redacted]
Assignment #2

1. Sets – Closure

When I first started considering this problem, I noticed how a closed set should almost be a mirror image, with one side being negative.



Note how each element is a mirror copy of what is on the other side of the line in the same position, only the signs change. After making this observation I really saw the value in developing a good sorting algorithm, as after a set is sorted like the one above, we can then just check does the first element + the last element = 0, if so we can then check the second element and the second last element…etc. If all elements add with the elements on the other side of the "mirror" to equal zero we can then say the set is closed. I believe this is the most elegant solution we could have for this problem.



Elements on either side should add to equal 0, to be considered a closed set.

To sort the elements in the set I introduced my insertion sort algorithm. Please consult the beginning of this report for detailed background on insertion sort, the pseudocode behind it and how it works.

Extra Mile: Insertion Sort – By using insertion sort, the set outputted is nicely ordered for future use in other programs. Much more useful for the user and other programs.

Comparing elements on either side of the "mirror"

After my set had been sorted I then moved onto comparing elements on opposing sides of the set. Below you can find my pseudocode solution to this problem

```
;           while(aStartingAddress<=aChangingAddress)
;   {
;               testValue = Memory.word(aStartingAddress);
;               currentValue = Memory.word(aChangingAddress);
;               zeroNumberTest = currentValue - testValue;
;               if(zeroNumberTest = 0)
;               { //Branch inconditionally to notClosed }
;               valuesAdded = testValue + currentValue;
;               if(valuesAdded = 0)                 //If it isn't it branches to notClosed
;               {
;                   aStartingAddress += 4;
;                   aChandingAddress -= 4;
;           }                                       //Branches back to the top of while loop if valuesAdded = 0;
;               }                           //Curly Brackets placed this way to explain the branching present
;               else
;               {
;                   closed = false;
;                                           //branches to end of program
;               }
;       closed = true;
;   }
;}
```

Note how the condition in the while loop will stop after the starting address (called aStartingAddress) has become greater than the decreasing finished address (called aChangingAddress), during each loop I both increase the address in the starting address and decrease the address in aChangingAddress, this allows for new elements either side of the set to be checked, whilst stopping after all the elements on one side of the set have been checked with the other side.

Extra Mile - Zeroes

At the start of this loop it loads a testValue and a currentValue, it then checks if those elements subtracted from each other equals zero. If it is, it exits the loop and declares this is not a closed set. As any negative number, or zero number minus another number cannot equal zero, it will only do so if there is more than one zero in the set, or the set is already invalid. This is also really just some error handling to make sure a user doesn't enter more than one zero.

After this, it then checks if testValue and currentValue added together equal zero, if this is a closed set, testValue and currentValue added together should always equal zero. It then exits the loop if this is not true, or if it is true, it changes address and repeats the process loading two new values into testValue and currentValue.

After all elements have been sorted, the program will leave the loop. It will then set the Boolean (R0) to either true (1) or false (0), depending on what the program did within the comparing loop.

Trinity College Dublin
The University of Dublin

School of Computer Science and Statistics
CS1021 - Introduction to Computing I
Conor Gildea - [redacted]
Assignment #2

Methodology for "Sets – Closure"

Inputted set:	**Outputted Set:**	**Result in R0:**	**Correct/Wrong – Why?**
Asize: 9 **AElems:** {+4,-6,-4,+3,**0**,-8,+6,+8,-3}	**Asize**:9 **AElems:** {-8,-6,-4,-3,**0**,3,4,6,8}	**R0:** 1	**Correct** (R0=1) This is a closed set and the program computed this correctly. Note how it also works with a zero in the set. We can also see the benefit of sorting as the output is fully sorted.
Asize: 9 **AElems:** {+4,-6,-4,+3,**0**,-8,+6,+8,-3}	**Asize**:9 **AElems:** {-8,-6,-4,-3,**1**,3,4,6,8}	**R0:** 0	**Correct** (R0=0) This is not a closed set and the program computed this correctly. Note how I replaced the "0" in the previous inputted set with a "1". Yet again we can see the benefit of sorting the output.
Asize: 0 **AElems:** {}	**Asize**:0 **AElems:** {}	**R0:** 0	**Correct** (R0 = 0) When the size of set A is entered as "0", the program should state this is not a closed set which this program does. Based on my research for this assignment, one of the key characteristics of open sets is that the intersection of any two open sets is open. If the empty sets were not considered open, then that wouldn't be true anymore. Thus empty sets should not be considered closed. This the program is correct.

Asize: 4 **AElems:** {1,-1,**"g","-g"}**	**Asize:** 4 **AElems:** {1,-1,**"g","-g"}** *Error message outputted* "Closure.s(93): error: A1116E: String operands can only be specified for DCB"	**N/A**	**Limitation of Language** For this test I wanted to see would it be possible for this program to work with ASCII characters. Due to language limitations the program was not able to sort and compute was this set, of numbers and letters (possibly could be interpreted by a user as variables), a closed set, as it was the incorrect data type. Possibly in a higher level language this would have been more feasible.
Asize: 4 **AElems:** {0,0,0,0}	**Asize:** 4 **AElems:** {0,0,0,0}	**R0:** 0	**Correct** (R0 = 0) As zero can only appear once in a closed set, this program computed the correct answer.
Asize: 4 **AElems:** {7,3,2,4}	**Asize:** 4 **AElems:** {2,3,4,7}	**R0:** 0	**Correct** (R0 = 0) As all these numbers in this set are positive numbers, this is not a closed set. This program computed the correct result whilst sorting the set.

2. Sets – Symmetric Difference

For this problem, my solution begins by first sorting the elements in A and the elements in B. Please consult the beginning of this report for how this was achieved and I implemented. To sort different elements, I simply inserted different values into the necessary registries and branched into the correct location. If we were using a higher level language such as Java I could have had a function saved would have allowed me to simply call it, instead of just branching.

After Set A and Set B have been sorted

I then compared the sets, using the following pseudocode to see what elements intersected between set A and set B.

The following pseudocode is a series of if statements however it does also loop back using branch statements in two locations however for clarity in the pseudocode I used if statements, with comments around branch points, as a single while loop would be too complicated for an anyone to follow easily.

```
if (finished != true)
{
   changingBAddress = inputtedBStartingAddress;
   changingAAddress = inputtedAStartingAddress;
   sizeBAddress = inputtedBSizeAddress;
   sizeB = Memory.word[sizeBAddress];
   sizeAAddress = inputtedASizeAddress;
   sizeA = Memory.word[sizeAAddress];
   countA = 0;
   countB = 0;

   if(sizeA!=0)||(sizeB!=0)
   {


       if(sizeA!=countA)
       {
           if(sizeB!=countB)
           {
               testBValue = Memory.word[changingBAddress];
               testAValue = Memory.word[changingAAddress];
               if(testBValue==testAValue)
               {
                   mask = mostSignificantBit(1)
                   testBValue += mask;  //This should act like addition
                   testAValue += mask;  //in most cases
                   Memory.byte(insertionCheckingAddress) = testValue;
                   Memory.byte(insertionCheckingAddress) = testValue;
               }
               changingAAddress += 4;
               countA++;
       //Branches unconditionally back to whileCheckingIntersection
       }                                                  //Curly Brackets placed this way to explain the branching present
           changingAAddress = inputtedAStartingAddress;
           changingBAddress += 4;
           countB++;
           countA = 0;
       //Branches unconditionally back to whileCheckingIntersection
           }
```

This pseudocode works by doing the following:

It gets the first element in set B, it then compares it will all the other elements in set A, if two elements match, we cross them off, by making the most significant bit convert to a

1 in each number. By converting the most significant bit of each number in each set that are equal, we can then reconvert it later to get back the original number.

As this problem is only meant to work with unsigned numbers, I believe this is a worthy trade off. It prevents us from working with exceptionally large number as intersections in sets however it does mean we can "uncross" the numbers that are intersections later on, and we are able to get back their original value. Please consult limitations of my assignment at the end of this report for more detail.

Transferring Elements to C

After the elements that are intersections in set A and set B have been crossed out, we can then move the other elements from these sets into set C. We do this by simply having a loop which goes through and loads first value of a set, if that value's most significant bit is equal to 1, we don't transfer it, it is isn't we do transfer it.

For each number we compare we have a count that increases, once the count reaches the size of the set we quit the loop. We also have a count that measures each time a number is transferred to set C, so we can calculate the size of set C.

If a number has been transferred, we then increase the address we are loading for the next free space for elements in C. In each loop regardless of transfer we also increase the address to load the next number in set A/set B to try and transfer it. After set A was transferred I simply changed values in the registries then branched back into this transferring loop, to transfer the values of set B.

<u>Extra Mile: Uncross elements in A and B</u>

Uncrossing the elements in set A and B is much like the above solution to transferring elements in set A and set B to set C. The only real main difference is that once I compare is an element from set A/set B's most significant bit equal to 1, I then use exclusive or and a mask on that element to turn the element's most significant bit back to 0. I then simply save this element in its normal address in set A/set B.

We can see this in pseudocode form, below.

```
testAddress = inputtedBStartingAddress;
if (changingAddress<testAddress)
{
   setElementCount = 0;
   changingAddress = inputtedAStartingAddress;
   sizeAAddress = inputtedASizeAddress;
   sizeA = Memory.word[sizeAAddress];

   while(sizeA!=setElementCount)
   {
       possibleCrossed = Memory.word[changingAddress]
       if(possibleCrossed's most significant bit is equal to 1)
   {
           mask = mostSignificantBit(1)
           possibleUncross - mask //In most cases this exclusive or will act as subtract
           Memory.word(changingCAddress) = possibleCrossed;
       }
       changingAddress += 4;
       setElementCount++;
//Branch unconditionally to unCross
   }
```

Finding CSize

We also must simply load the address for CSize and store the value of the counter we made, during the transfer of elements, which increased each time an element was transferred to C. This allows us to have a correct value for CSize, given the fact this was an unknown originally. After CSize is found we then sort the elements in C and the program is finished.

Extra Mile – All Three sets sorted – In this solution I sorted all three sets into their correct order, this is much more pleasant for the user and easier for other programs to work with these sets in the future.

Extra Mile – Regained the real value of crossed out elements – I was able to uncross the insertion values of each set after the correct values had been transferred to set C. This allowed for the user to be able to see all their original elements they had in each set, for future use.

Trinity College Dublin
The University of Dublin

School of Computer Science and Statistics
CS1021 - Introduction to Computing I
Conor Gildea - [redacted]
Assignment #2

Methodology for "Sets – Symmetric Difference"

Inputted Sets:	Outputted Set:	Correct/Wrong – Why?
ASize: 8 **AElems:** {4,6,2,13,19,7,1,3} **BSize:** 6 **BElems:** {13,9,1,20,5,8}	**ASize:** 8 **AElems:** {**1**,2,3,4,6,7,**13**,19} **BSize:** 6 **BElems:** {**1**,5,8,9,**13**,20} **CSize:** 10 **CElems:** {2,3,4,5,6,7,8,9,19,20}	**Correct** -Correct Elements in C -Correct CSize for C Note the bolded numbers in set A and set B, these were the intersections. From the output for CElems, we can clearly see it contains the correct symmetric difference, as none of the intersection numbers have been included in set C. Thus this program computed this test case correctly.
ASize: 6 **AElems:** {4,6,2,19,7,3} **BSize:** 4 **BElems:** {9,20,5,8}	**ASize:** 6 **AElems:** {2,3,4,6,7,19} **BSize:** 4 **BElems:** {5,8,9,20} **CSize:** 10 **CElems:** {2,3,4,5,6,7,8,9,19,20}	**Correct** -Correct Elements in C -Correct CSize for C Note for this example I removed all the numbers that were intersections in the previous example, just to see what the result would be. The output for C remained the same. All elements in C only appeared once in either A or B. Thus this program computed this test case correctly.

ASize: 6 **AElems:** {4,6,2,19,7,3} **BSize:** 6 **BElems:** {2,3,4,6,7,19}	**ASize:** 6 **AElems:** {2,3,4,6,7,19} **BSize:** 6 **BElems:** {2,3,4,6,7,19} **CSize:** 0 **CElems:** {}	**Correct** -Correct Elements in C -Correct CSize for C Note for this example I purposely kept all the elements from set A in the previous example the same. I also transferred those elements to set B. If this program is working correctly, there would be no elements in C and the Size would be 0. From the outputted sets we can see that this program has CSize as 0 and there are no elements in set C. Thus this program computed this test case correctly.
ASize: 6 **AElems:** {4,6,2,19,7,3} **BSize:** 0 **AElems:** {}	**ASize:** 6 **AElems:** {2,3,4,6,7,19} **BSize:** 0 **AElems:** {} **CSize:** 6 **CElems:** {2,3,4,6,7,19}	**Correct** -Correct Elements in C -Correct CSize for C Note for this example I purposely kept all the elements from set A in the previous example the same. I also made set B an empty set with a size of zero. If this program is working correctly, only all the elements in set A should be in set C and the size of set A should be the same size as set C. From the outputted sets we can see that this program has CSize as 6 and the elements in C are the same as the elements in A.

		Thus this program computed this test case correctly.
ASize: 0 **AElems:** {} **BSize:** 0 **AElems:** {}	**ASize:** 0 **AElems:** {} **BSize:** 0 **AElems:** {} **CSize:** 0 **CElems:** {}	**Correct** -Correct Elements in C -Correct CSize for C Note for this example I made both set A and set B empty sets. If this program is working correctly, set C should be of size 0 and contain no elements. From the outputted sets we can see that this program has CSize as 0 and there are no elements in C. Thus this program computed this test case correctly.

3. **Anagrams**

I felt the final problem anagrams was especially non-trivial after I had prepared my insertion sort algorithm for the other parts. What I found interesting during this problem, was that it wouldn't have been that much easier to code the solution to this problem in a higher level language, as we are working with such small elements, just characters.

As no size was provided this time for the sorting algorithm, I coded the program to calculate how many elements were in each string, this was done by having a loop that loaded the first value in the starting address of each string, it would then check was the value of that element null, it wasn't it would continue to loop and a counter was increased.

Extra Mile – Uppercase scenarios handled

Whilst loading the characters separately in each string in my counting loop, I also used this opportunity to make sure all characters were in lowercase not uppercase, to keep it uniform and make them easy to compare.

Extra Mile – Other inputs rejected

After I converted uppercase letters to lowercase letters, I then rejected all characters that were not lowercase letters. As anagrams are only meant to consist of letters this step was necessary to prevent the user from entering non-letter characters.

After the counting loop was null terminated, I then had access to the size of the string in stringA and string B and I used this in my sorting algorithm, to prevent it sorting beyond the amount of elements in each string.

I then simply put both strings and their sizes separately through my insertion sort algorithm, this time the address increment was 1 instead of 4 as we were checking characters in the string. Once these strings were sorted the process was very simple to check was this an anagram.

After Insertion Sort

After both strings have been sorted, this problem was very simple. I compared the first character of stringA with the first character of stringB and continued checking until both null terminated at the same time, or the characters were not equal to each other and this wasn't and anagram. If it wasn't an anagram, the anagram Boolean (R0) would be set to 0, it was, it would be set to 1.

Please consult my comparing characters pseudocode on the following page for reference.

Comparing Characters Psuedocode

```
charA = Memory.byte(stringAAddress)
charB = Memory.byte(stringBAddress)
if(charB = NULL)
{
   if(charA=NULL)
   //Branches unconditionally to finishedComparingAll - Out of whileComparing loop
}
if(charA=charB)
{
   stringAAddress++;
   stringBAddress++;
   //Branches unconditionally to whileComparing - Continuing the loop
}
```

Extra Mile – Different size strings do not display as anagrams even if they contain the same elements, with one being larger. **E.G. "coats" and "taco"**

In my program both elements must null terminate at the same time and have all the same characters to be considered an anagram, this error handling prevents the incorrect outcome being displayed.

Trinity College Dublin
The University of Dublin

School of Computer Science and Statistics
CS1021 - Introduction to Computing I
Conor Gildea - [redacted]
Assignment #2

Methodology for "Anagrams"

Inputted Strings:	Outputted Strings:	Output in R0:	Correct/Wrong – Why?
StringA: "coats" **StringB:** "tacos"	**StringA:** "acost" **StringB:** "acost"	**R0:** 1	**Correct** The program correctly identified that "coats" was an anagram of "tacos"
StringA: "COATs" **StringB:** "tacos"	**StringA:** "acost" **StringB:** "acost"	**R0:** 1	**Correct** The program correctly identified that "COATs" was still an anagram of "tacos", the program was correctly able to handle uppercase letters in the string.
StringA: "" **StringB:** ""	**StringA:** "" **StringB:** ""	**R0:** 0	**Correct** The program correctly stated that when there are two null sets there are no anagrams present. I believe this is entirely the right output to present, as you cannot have an anagram of nothing.
StringA: "tacos-" **StringB:** "cOat.s"	**StringA:** "tacos-" **StringB:** "cOat.s"	**R0:** 0	**Correct** As anagrams consist of letters in a word being used to create a different word this program rejects when there is a character in the string that isn't a letter. Anagrams must contain just letters. Thus this program computers valid output when it states R0=0. This error handling makes sure users only enter in letters.

Breakdown of My Assignment:

1. Sets - Closure:

Can compare numbers in a set to verify is it a closed set or not.

E.g. {-3,0,3} is closed

{-1,-2,0,3} is not closed

Extra Mile: Works perfectly even with zeros

Extra Mile: Worked with null sets

Extra Mile: Insertion Sort – Provides much friendlier output set for use in other programs.

2. Sets – Symmetric Difference:

Computed the correct size for set C

Transferred the elements that were not an intersection of A and B, into C

Extra Mile – Insertion Sort A, B and C - Provides much friendlier output set for use in other programs

Extra Mile: Worked with null sets

Extra Mile – Program functions properly even when both or either set is null.

Extra Mile – Regained the real value of crossed out elements

3. Anagrams:

Checks two strings to see are they anagrams of each other.

Extra Mile – Different sizes of strings do not display as anagrams even if they contain the same elements, with one being larger. E.G. "coats" and "taco"

Extra Mile: Worked with null strings.

Extra Mile – Uppercase scenarios handled and converted to lowercase.

Extra Mile – Other inputs rejected – Error handling – Prevents the user from behaving badly. Also anagrams are only meant to consist of letters being mixed from one word to create a new word, anagrams do not consist of other characters that are not letters.

Limitations of my Assignment:

In **Sets – Symmetric Difference**, I cross out the numbers that are intersections of set A and B by changing the numbers most significant bit to 1, then moving all elements (from set A and set B) besides the elements with their most significant bit equal to 1, into set C. I then convert the crossed elements back into their original form by using exclusive or. By doing this it does enable me to get the crossed numbers back into their original form but it does mean really large numbers may not be able to compute properly if they are intersections of set A and B.

However I believe this is worth the trade off as we were told we are using unsigned values for this problem, not signed values. It is more likely to benefit the user by displaying the crossed out numbers back in their original form, than it is for the user to be able to enter exceptionally large unsigned values that happen to be intersections of both sets A and B. If I wanted to I could have allocated more area in memory and stored all intersection numbers there however I believe this would been unnecessary unless we knew the user was working with exceptionally high elements in their sets.

In **Anagrams**, I sort both the characters in both strings alphabetically, this means the strings are likely not in their original order at the end of the program, however I believe this is worth the trade off, for the elegant solution to this problem. Also it was not required to keep both strings, in their original order, so it shouldn't matter too much anyways.