` Assignment 2

Sets-Closure

Requirement:

To determine if a set is closed under negation and update a Boolean register to represent if it's closed or not.

Assumptions:

The set does not repeat elements. Only 32 bit signed integers.

Logic behind design:

- Check if element at address is 0. If 0 go to next address.
- Else add element to next element of set to see if it equals 0
- If 0 set both to 0 and repeat from step 1. Else repeat step 2.
- If the element is never negated, then not closed.
- If every address is checked then it is closed

Table of test sets and results:

Set	**R0(Boolean)**
+4,-6,-4,+3,-8,+6,+8,-3	1
+4,-6,-4,+3,-8,+6,+8,-5	0
+4,-6,-4,+3,0,-8,+6,+8,-3	1
+4,-6,-4,+3,0,-8,+6,+8,-5	0
NULL	1
0	1
+9	0
2147483647,- 2147483647	1

Advantages and disadvantages of method:

The main advantage of this method is simplicity and relative efficiency. There is only one condition under which it is closed and only one condition under which it isn't. This makes the algorithm easy to follow and modify. Also the algorithm does not have any conditions under which it doesn't work or requires a different way to calculate negation.

One disadvantage is that this method 'eats' both the count and the values in the set. This was a sacrifice made for efficiency so as not to make a copy of the set and count and not have to somehow remember which elements have been negated so far.

Also if 0 is not a part of the initial set, it would be faster to check if the count is uneven or even. Then if it is even you only need to check half the non-zero values in the set for negation. However, since 0 was a potential member of the sets the solution would require checks for uneven etc. Using very simple

heuristics I determined that the end efficiency for both would probably end up being very similar as a result.

Sets-Symmetric Difference

Requirement:

Move only one copy of each element from both sets to a new set C.

Assumptions:

Each individual set does not repeat elements

Logic behind design:

- Copy all of one set into set C (as sets can't have duplicate elements this is fine)
- Compare each element of other set against every element in set C.
- When match is found compare next element in other set. If no match found, add to set C and increment number of elements in C.
- End when all elements of other set is checked

Table of test sets and results:

Set A elements	**Set A count**	**Set B elements**	**Set B count**	**Set C elements**	**Set C count**
4,6,2,13,19,7,1,3	8	13,9,1,20,5,8	6	4,6,2,13,19,7,1,3,9,20,5,8	12
null	0	1	1	5	1
null	0	null	0	null	0
1	1	5	1	1,5	2

Advantages and disadvantages of method:

The algorithm itself seems efficient as far as I can tell. However, in my implementation one potential source of improvement would be to somehow not bother checking an element that already had a match. This is because in each individual set all the values are unique, hence if a member of set B matches a member in set C there is no reason to check any other member in set B against that member in C.

Anagrams

Requirement:

Check if two null terminated strings in memory are anagrams of each other and update a Boolean register to represent if they are or not

Assumptions:

Strings contain only ASCII characters

Logic behind design:

- Clear a space in memory to store the frequencies of each letter of alphabet
- Begin processing string A
- Check each letter in string to see if uppercase, lowercase or symbol
- If letter increment the frequency counter in appropriate memory location
- Move to next string and repeat last two steps, however decrement the counter instead of increasing it.
- Once both strings are processed check if every frequency is 0. If 0 it is anagram, if not it's not anagram

Table of test strings and results:

String A	**String B**	**R0**
coat	taco	1
better	bet	0
Merry Christmas	CHRIST MAS,MERRY	1
AnAgRaM	anagram	1
“”(empty)	“”	1
beets	bests	0
“”	holiday	0

Advantages and disadvantages of method:

There are a couple of clear disadvantages, however this specific implementation is intentionally geared towards short strings like single words, rather than sentences or words. However, as I will explain soon, this is an easily rectifiable problem and was an intentional choice rather than drawback to the algorithm. Two advantages of this method are that it does not modify the original strings and it only parses over each character of each string once.

The reason this method is geared towards small strings is because it can only store unsigned byte sized frequencies. This means if any letter repeats over 16 times it could cause overflow. Technically as this method uses unsigned frequencies it only cares about relative difference in frequencies. This means it technically works for infinite sized strings as long as no alphabet appears in one string 16 or some multiple of 16 times more than the other.

This method however, despite being implemented to work for small strings can easily be extended to work for larger strings. We simply need to use half-words or words instead of bytes to store the frequencies. This comes with the obvious drawback of using more memory and possibly being less efficient.

Also for clarity and ease of understanding program flow, I decided to use two separate registers to store string memory. This is unnecessary as we can re-use the same register.

Another noticeable drawback is the initial overhead in assigning memory. This is unavoidable with this method so for two or three letter words this method may be relatively inefficient. However, the advantage this method offers in return is that it only parses over each character once thereby cutting down on instructions in the long run for larger strings.

A slight tradeoff for efficiency could be made for this program. As this program is geared towards words rather than sentences there is little need to check for symbols. Therefore, rather than using 4 CMP instructions we can simply CMP against ‘a’ and treat it as uppercase if lower and lowercase if greater or equal. The decision to explicitly check for symbols is to make it easily scalable for larger strings.