1. We are given $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_n$ and z. The first step is to reformulate the problem slightly. To do this, let $w_i = z - y_i$. Then,

$$z \in \{x_i + y_i \mid 1 \le i, j \le n\} \quad \textit{iff} \quad x_i = w_j \ \textit{for some} \ 1 \le i, j \le n$$

This suggests that we create two hash tables, T_x and T_w, each with n entries, and each using the same hash function, h. We then hash each of the x_i to table T_x, and hash each of the w_i to table T_w. If $x_i = w_j$, then both x_i and w_j will hash to the same bucket of their respective tables. Thus, we need only examine the buckets of T_x and T_w. Specifically, for each k, we compare bucket k of T_x with bucket k of T_w. If they contain a common value, then we return `true`. If no such pair of buckets contains a common value, then we return `false`. The algorithm below makes this idea precise, where H is a universal set of hash functions from U to $[0, n-1]$. The expected size of each bucket is constant, so it is not hard to show that the entire procedure takes $O(n)$ time.

```
ALGORITHM:
     create a hash table Tx with n buckets
     create a hash table Tw with n buckets
     randomly choose a hash function h from H

(1)  for i from 0 to n-1 do
       store xi in position h(xi) in table Tx
       end

(2)  for j from 0 to n-1 do
       store z-yj in position h(z-yj) in table Tw
       end

(3)  for k from 0 to n-1 do
       for each item x in bucket k of Tx do
         for each item w in bucket k of Tw do
           if x=w then return true
           end
         end
       end

     return false
```

By Theorem 12.3 in the text, the expected length of a chain attached to a bucket in table T_x is at most 1, since n items are being stored in a hash table with n buckets. Likewise for the chains in table T_w. Thus, the expected time required for each iteration of loop (1) is bounded by a constant, c_1; and the expected time required for each iteration of loop (2) is bounded by a constant, c_2; and the expected time required for each iteration of loop (3) is bounded by a constant, c_3. (In the case of loop (3), the expected time required for complete execution of the inner two loops is bounded by a constant.) Thus, each of loops (1), (2) and (3) takes expected time $O(n)$, so the entire algorithm takes expected time $O(n)$.

2. (a) What is the worst-case time complexity of a single operation in a sequence of m EnQueue and DeQueue operations? Derive matching upper and lower bounds. That is, define an initial

situation by describing what H and T look like at the start, and then define a sequence of m operations, where the sequence consists of EnQueue's and DeQueue's. Then show that one of the operations in the sequence (probably the last operation) will have the claimed worst-case time. For the upper bound, show that no operation in any m-operation sequence can ever take more time than the claimed worst-case time.

Solution:

An EnQueue operation always takes $\Theta(1)$ time. DeQueue takes $\Theta(1)$ time if the stack H is nonempty, and takes time proportional to the size of the stack T if H is empty. Hence, the most time-consuming operation is a DeQueue applied when H is empty, and T has as many elements as possible. Since both H and T are initially empty, after m operations we can at most hope to have m elements in T, so that the most expensive operation will take $O(m)$ time. We will now show how to construct a sequence of operations so that the cost of the most expensive operation is $\Theta(m)$.

We choose the first $m-1$ operations to be EnQueues. At this point, H is empty and T contains $m-1$ elements. The mth operation is chosen as a DeQueue. By the preceding discussion this will take $\Theta(m)$ time.

(b) Use the accounting method to prove that the amortised time complexity of each operation in a sequence of m EnQueue and DeQueue operations is $O(1)$.

To solve this problem, first give a credit scheme indicating how many credits to allocate to each EnQueue and DeQueue operataion. Secondly, state the credit invariant, and thirdly, prove the credit invariant.

Solution:

Our credit scheme is to allocate 4 credits to each EnQueue, and 2 credits to each DeQueue. The credit invariant is that each element of T contains 3 credits.

Initially, T is empty, the credit invariant is true.

When we perform an EnQueue, we use one of the credits to pay for pushing the element into T, and store the remaining three credits with the element in T. This ensures that that the credit invariant remains true.

When we perform a DeQueue we use one credit to pay for testing whether H is empty, and the other for popping the top element of H. There remains to account for the cost of transferring the elements of T to H, in the event H is empty. We need 3 credits to transfer each element from T to H (one for testing if T is empty, one for popping the element from T, and one for pushing it into H). To pay for this, we use the 3 credits stored with the element in T.

Since we have an adequate credit scheme that uses $O(1)$ credits per operation, the amortised cost of each operation is $O(1)$.

3. Recall that the doubling method enables the implementation of a stack without placing a limit on the size of the stack, such that the amortized complexity of each operation is $O(1)$. Every time the array gets full, a new array is allocated whose size is twice the size of the old array, and the old array is copied to the new array.

(a) Suppose we change the implementation so that the size of the new array is 3/2 times the size of the old array. What is the time complexity of a sequence of m operations in the worst-case? Justify your answer.

The time complexity should still be $O(1)$ per operation. However, we are copying more often, so the charge (and amortized cost) should be greater. Let's try to charge 4.00 dollars, one dollar to put the item in the array, one item to copy, and two dollars leftover for future copies

for neighbors to our left. The credit invariant is that every item in the last (rightmost) 1/3 of the array has 3 dollars stored, one dollar to pay for its own copy, and 2 left to pay for *two* neighbors to the left. Also at every point in time, the array is between 2/3 full, and completely full. The proof is by induction. Base case is vacuously true since nothing is stored. For the inductive step, there are two cases. The first case is where we do an append without overflow. In this case, we use one dollar to pay for append, and the remaining 3 are stored as credit, so the credit variant is maintainted. The second case is where we do an append with overflow. In this case, before the append, if the array size is n, then we have $3 \times (1/3)n = n$ dollars in the bank. We use this stored amount to pay for all copies, and then 1 of the 4 dollars charged to pay for putting the new item in the array, and the remaining 3 as credit.

(b) Suppose we change the implementation so that the size of the new array is 50 plus the size of the old array. What is the time complexity of a sequence of m operations in the worst-case? Justify your answer.

This will not work, since every 50 steps, we are doing a full copy. Let $T(n)$ be the time to complete the $50n$. Then we have $T(1) = 50$, and $T(n) = T(n-1) + (n-1)50 + 50 = T(n-1) + 50n$. This is because it takes $T(n-1)$ time steps for the first $50(n-1)$ copies, and then for the last 50 copies, we first have to move all $50(n-1)$ old items to a new array, and then move the new 50 items to the new array. Solving this gives $50(1 + 2 + \ldots + n) = \Theta(n^2)$.