TASK 1

1.1) This algorithm cycles through array B and searches for the value that is equal to the key value. It returns the index of this value in A (j + 2*i), based on the position inside B (i), or returns "-1" if there is no match. We can name input j – "the index of the sub-array", and because the original array has been separated into two sub-arrays based on the modulo of 2 (even or odd), the input j refers to (the index of the value in A) mod 2 (i.e., for the sub-array A1 j = 0, and for the A2 j = 1).

1.2) function (RecursiveFind(key, B, M, j, I), we start with i = 0

```
if(i >= M):
        return -1
if(B[i] == key):
        return j + 2 * i
return RecursiveFind(key, B, M, j, i + 1)
```

1.3) The master theorem is used to compute the time for "divide and conquer" algorithms, and, based on the fact that the problem is split into smaller subproblems, these subproblems are solved recursively and, then solutions are combined to give a solution to the original problem. But the recursive find is more "decrease and conquer" algorithm: we don't split our problem and recombine solutions, but rather reduce it to a single smaller problem. Because there are no splitting/recombining solutions - we don't need the master theorem to calculate the computing time. We also can reason this out logically: RecursiveFind is a search algorithm that is based on comparisons (we don't have information about the original array being sorted), therefore if there no key value in the array (worst-case scenario), we need to look up all values inside the A.

TASK 2

Function - Line of code	**Best-case input** **(key, A1[key...], A2, N)** For the best-case input, we assume that the first value of the A1 is equal to the key value, but the values themselves don't matter	**Worst-case input** **(key, A1[no key], A2[no key], N)** For the worst-case input, we assume that there is no key value either in A1 nor in A2
R0 – Line 1 (beside function calls)	Constant run time: C0	
Ceiling (assume constant runtime that doesn't depend on argument)	Constant run time: C1	
Find – Line 1 Find – Line 2 Find – Line 3 Find – Line 4	Because A1[0] = key the loop will iterate only once and return 0. Lines 1-3 will all have constant run times – C2, C3, C4, so the function "Find" has run time: C5 = C2 + C3 + C4 and returns the value 0	Because there is no key value in the array A1, then line 1 will run Ceiling (N/2) + 1 times, line 2 – Ceiling (N/2), and line 3 will never fire up. Because the value of Ceiling(N/2) depends on N in a linear fashion, we can establish that the "Find" function has runtime: T(N) = C2*N + C3 and returns -1
R0 – Line 2 R0 – Line 3 (beside function calls)	Condition check has constant run time: C6, and because index = 0, line 3 will be skipped	Constant run time: C4
Floor (same as Ceiling)	Skipped (line 2 of R0 – false)	Constant run time: C5
Find – Line 1 Find – Line 2 Find – Line 3 Find – Line 4	Skipped (line 2 of R0 – false)	The same principle as the previous "Find" call, runtime T(N) = C6*N + C7 and returns -1
R0 – Line 4	Constant run time: C7	Skipped (line 3 of R0)
TOTAL RUN TIME	Best-case input has run time: T(N) = C0 + C1 + C5 + C6 + C7 = C8	The worst-case input run time: T(N) = C0 + C1 + C2*N + C3 + C4 + C5 + C6*N + C7 = (C2 + C6)*N + (C0+C1+C3+C4+C5+C7) = C8*N + C9

The growth function	1 (Because run-time is constant)	N
The Theta notation	$T(N) = C8 = \Theta(N^0) = \Theta(1)$ $c1 = \{x \in R+ \mid x \in (0;C8]\}$ $c2 = \{y \in R+ \mid y \in [C8;+\infty)\}$ $m0 \in \{z \in R+ \mid z \in (0;+\infty)\}$ $(0; C8] \leq C8 \leq [C8;+\infty)$ for all $N > 0$	$T(N) = C8 * N + C9 = \Theta(N)$ $c1 = \{x \in R+ \mid x \in (0;C8]\}$ $c2 = \{y \in R+ \mid y \in [C8 + C9;+\infty)\}$ $m0 \in \{z \in R+ \mid z \in [1;+\infty)\}$ $(0; C8*N] \leq C8*N + C9 \leq [C8*N + C9*N; +\infty)$ for all $N \geq 1$

TASK 3

3.1)
```
function R1(key, A, B, N)
    ind = -1
    for (0 ≤ i < N):
        if(A[i] =/= B[i]):
            return -2
        if(A[i] == key AND ind == -1):
            ind = i
    return ind
```

Lines 1-2: Initialize the variable 'ind' that stores the resulting index and start iterating over arrays
Lines 3-4: We check for the hardware malfunctions and return -2 if there is a data error
Lines 5-6: If data hasn't been altered, we check if this value in both arrays is equal to the key value. If it is, and it is the first instance – we set the 'ind' variable.
Line 7: We return the result of the search: '-1' if the value cannot be found, or the index of the key value is stored in both arrays.

3.2) As we can see from the form of the algorithm: the worst-case input is two identical arrays that store the "key" value somewhere. In this case, the algorithm will have to cycle through 2 loops of length N and return -1 because data is OK, but the "key" value is missing. Therefore, if we translate "for ($0 \le i \le N$)" loop as {i = 0 ; if (i < N): … ; i = i + 1} and count each simple operation as taking 1 time unit, then: line 1 has a runtime of 1, the loop without body will has a runtime of 7N + 5. Condition on line 3 has a run-time of 5 and will fire N times => 5N, while line 4 will never execute 1*0 = 0. Line 5 takes 8 time units and runs N times => 8N, and Lines 6 and 7 both fire only once and take – 2 time units.
The resulting runtime for the worst-case input of the R1 is T(N) = 1 + (7N+5) + 5N + 0 + 8N + 2 + 2 = 20N + 10, and $T(N) = \Theta(N)$, because for $n0 \in [10;+\infty)$, $c1 \in (0;20]$, $c2 \in [21;+\infty)$ => $(0; 20N] \le 20N + 10 \le [21N;+\infty)$ for all $N \ge 10$.

TASK 4

4.1)
```
function R1Hash(key, a, b, A, B, N)
    Hash ← new array(N^2) of '-1'-s
    ind = -1
    for (0 ≤ i < N):
        h = (a * A[i] + b) % N^2
        for (0 ≤ j < N^2):
            m = (h + j)%(N^2)
            if (Hash[m] == -1):
                if(B[m] != i):
                    return -2
                Hash[m] = 0
                break
        if(A[i] == key AND ind == -1):
            ind = i
    return ind
```

4.2) We declare and initialize the "hash" array and index variable, then loop through the array "A" to check the data integrity and look for the "key" value while we at it. Using the "hash" array as an auxiliary tool, we check if data array "B" stores indexes where it should, based on the arguments(values of "A", a, b), and if there is a deviation in data stored, the algorithm will report an error and return -2. If we confirmed that index 'i' is stored correctly, we then also check if the value A[i] is equal to the "key" value, and if it is, we change the index variable (only if it hasn't been changed already) value to "i". If we loop through the array "A" without data issues, the function will return the index of the "key" value or '-1' if there is no such a value. It should be noted, that intuition dictates that we use the fact that each index is stored in a specific location, and use direct access (return B[(a*A[i]+b)%(n^2)]) to find it instead of linear search. But the fact, that this algorithm uses linear probing for collisions makes direct access useless (example input: key = 2, a = 1 (>0), b = 0, A = [1, 1, 1], B = [-1, 0, 1, 2, -1, -1, -1, -1, -1], n = 3).

TASK5

I was trying to make an optimal storage-search algorithm: that is fast, would be usable even for colliding data, will offer at least basic data security, and will not take too much space. We can modify this algorithm for better data storage and increase the number of data arrays passed as arguments, and, in case of data errors, we'll have a chance to use a backup array to restore the data. The array B is a two-dimensional array that stores N buckets of size N each. This way, we can resolve data collisions by sorting indices into smaller groups. Here I assumed that hash sum check will not suffice because it's not 100% fail-proof, so we have to check each value separately (one loop of size N). But, if we ever separate the search task (confirm the data once, then search it multiple times) then, we can find the key-value even faster just by searching the bucket. If values in the array are distributed uniformly, then the number of non-negative indexes in the bucket is less than N (while Bucket[i] != -1), and this way the search can be even faster.

```
function RSearch(key, A, B, N, a, b)
    arr ← new array(N) of '0'-s
    ind = -1
    for (0 ≤ i < N):
        h = (a*A[i] + b) % N
        if (B[h][arr[h]] != i):
            return -2
        arr[h] = arr[h] + 1
        if(A[i] == key AND ind == -1):
            ind = i
    return ind
```

Lines 1-2: We create the auxiliary array that we will use to check the current indexes of the buckets and variable 'ind' that will store the result (the initial implication is that there is no key value in the array A).
Lines 3-4: Loop through the array A and for each value we determine the index of the bucket that should store the index 'i'.
Lines 5-6: Now, we can check if the index 'i' stored correctly (auxiliary array 'arr' stores position inside each bucket) in the array B, and if there is a difference the function will return '-2' to indicate the data error.
Line 7: If data is OK we just increment the index of the next available position inside the bucket [h]
Lines 8-9: We also check if the 'key' value is stored in array A, and if it is, and this is the first entry (the variable 'ind' hasn't been changed yet) - we store the index 'i' to return later.
Line 10: In the end, we return the value of the variable 'ind' that shows that value is not present (-1) or the correct position of the key-value inside the array A.

	Best-case input **(key, A[m...], B[...], N, a, b)**	**Worst-case input** **(key, A[no key], B[...], N, a, b)**
	In the best-case input the first value of the array A - 'm' is such that B[(a*m + b)%N][0] != 0, other values don't matter	The worst-case input is the one which has correct data and includes the key value

Line 1	We assume that initialization of the array of the size N is taking (C1*N + C2) time units to run	
Line 2	Constant run time: C3	
Line 3	Because this input will signal the error with data on the first iteration the loop will have constant run time: C4	For this input we will iterate over the loop N times without return, so the loop itself will have the run time: (C4*N + C5)
Line 4	Constant run time: C5	The lines themselves has constant run time, but the loop body will be fired N times. Line 6 will never execute, and the line 9 will execute only once. So, in total we can say that the loop body will take (C6* N + C7) time units to execute.
Line 5	Constant run time: C6	
Line 6	Constant run time: C7	
Line 7		
Line 8		
Line 9		
Line 10		Constant run time: C8
TOTAL RUN TIME	Best-case input has run time: $T(N) = (C1*N + C2) + C3 + C4 + C5 + C6 + C7 = C1* N + C8$.	The worst-case input run time: $T(N) = (C1*N + C2) + C3 + (C4*N + C5) + (C6*N + C7) + C8 = C9*N + C10$.
The growth function	N	N
The Theta notation	$T(N) = C1*N + C8 = \Theta(N)$ $c1 = \{x \in R+ \mid x \in (0;C1]\}$ $c2 = \{y \in R+ \mid y \in [C1 + C8;+\infty)\}$ $m0 \in \{z \in R+ \mid z \in [1;+\infty)\}$ $(0; C1*N] \leq C1*N + C8 \leq [C1*N + C8*N; +\infty)$ for all $N \geq 1$	$T(N) = C9*N + C10 = \Theta(N)$ $c1 = \{x \in R+ \mid x \in (0;C9]\}$ $c2 = \{y \in R+ \mid y \in [C9 + C10;+\infty)\}$ $m0 \in \{z \in R+ \mid z \in [1;+\infty)\}$ $(0; C9*N] \leq C9*N + C10 \leq [C9*N + C10*N; +\infty)$ for all $N \geq 1$