Algorithm and Data Structure 2 [001]

Task 1:

1. The best-case input: R0(7, A1, A2, Length of array A)

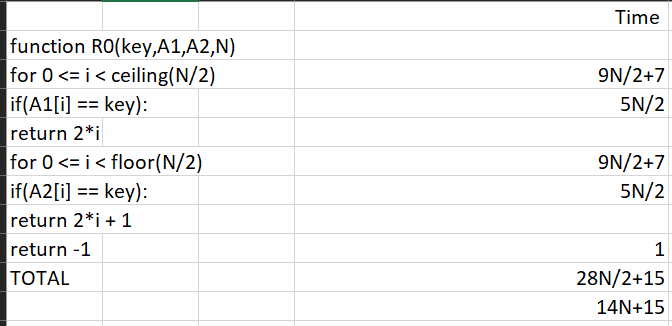
This will be the best-case input due to the position of comparison for array A1, the number 7 is the first array in A1 which is located in the first for loop, therefore we do not need to go through all element of the array A1 and A2

The first worst-case input: R0(10, A1, A2, Length of array A)

This input makes sure that it is the worst scenario as the number 10 is not inside any of the arrays, thus making the function goes through all commands.

The second worst-case input: R0(8, A1, A2, Length of array A)

This input has the same running time as if we can’t find the key. Finding the last element of A2 is also the worst-case because it will run through A1 and A2, the same as the first worst-case

1. 

This comes from the RAM model way of counting. We ignore the space that it takes because we only care for the time. each action cost 1 time unit, for example the if(A1[i] == key ) will take 5\*N time units, but our N is/2 so it becomes 5\*N/2. The 5-time units come from:

If takes 1

Calling the I take 1

CallingA1[i] take 1

Comparing take 1

Calling key take 1

And we multiply by N/2 because every iteration will do this again, so it becomes 5N/2

Therefore T(N) = 14N+15

1. The growth function of the running time is N. Because this algorithm growth depends on the size of the N. This can also be seen from the T(N), N is the highest degree on T(N)
2. T(N) is ϴ (g(N)) means that:

0 ≤ c1 · g(n) ≤ f(N)≤ c2 · g(n)

We can change the f(N) to our T(N)

0 ≤ c1 · g(n) ≤ 14N+15 ≤ c2 · g(n)

As the highest degree in T(N) is N, we can set g(n) to N.

0 ≤ c1 · N ≤ 14N+15 ≤ c2 · N

Here, we can set the lower bound to 14\*N and the upper bound to 15\*N+15.

0 ≤ 14N≤ 14N+15 ≤ 15N+15

This value will remain true for N > 0, therefore the **n0 is 1, c1 is 14 and c2 is 15**

Task 2:

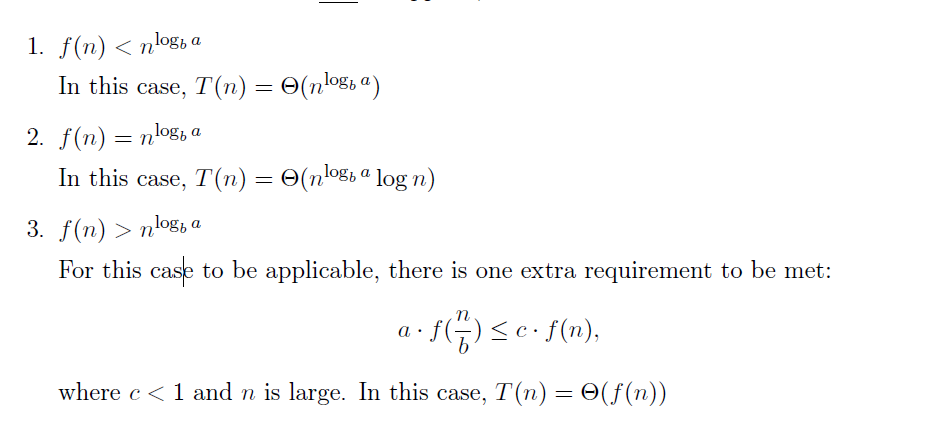
There may be a bug inside the pseudocode, in the sum algorithm, N is not defined and I assume that N is the length of array A

1. There are 2 inputs for the function createB, which is A (The array), and N (length of the array). The function will go through all of the array and find the sum of it; thus, it doesn’t matter the order of the array or the length of it, the best-case and worst-case will be the same as the function does not have any other stop condition and will just sum all of the element in the array.
2. First, we have to find the recurrence relation of it. The function Sum has 2 recursions, with each recursion halving the size of N. To solve it using the master’s theorem, we need to make a recurrence relation to this form:

T(N): aT(N/b) + f(N)

Because we have 2 recursion the a will be 2, this comes from the fact that we will execute 2 recursions, if we have 1 only the a will be 1. The b is the rate of our N decrease per recursion which is 2, because both of the recursion keep halving the N. Lastly, the f(N) is the work done outside the recursion, which in this case is f(1) or a constant because there is not a single loop or another recursion. Therefore:

T(N): 2T(N/2) + O(1)

Now we need to find which rule that we should use based on these:

First case:

O(1) < Nlogba

O(1) < Nlog22

O(1) < N

This is true, so the theta notation for this algorithm is based on the first rule which is T(N) = O(Nlog22), simplified to O(N)

Task 3:

1. Function R2(key,A,B,N):

key\_index = []

B = empty array

For 0 <= j < ceil(N/2):

If N%2 != 0 and i = ceil(N/2-1):

If A[N-1] = B[j]:

If key = A[N-1]:

Add N-1 to key\_index

elif (A[2\*j] + A[2\*j+1]) != B[j]:

return -2

elif (A[2\*j] + A[2\*j+1]) != B[j]:

return -2

elif (A[2\*j] + A[2\*j+1]) = B[j]:

if A[2\*j] = key:

add 2\*j to key\_index

if A[2\*j+1] = key:

add 2\*j+1 to key\_index

if length of key\_index = 0:

return -1

else:

return key\_index

I use array to store the index of key matching because there could be multiple matching items.

1. This algorithm’s worst-case is when there is no error inside the array A and B, for example A = [1,2,3,4,5,6,7,8] and B = [3,7,11,13] and the key is 8. The worst-case is when the algorithm executes all of its code. By looking at the algorithm, there exist only 1 loop which is the for loop that loops from 0 to ceil(N/2). Inside the loop, there are multiple ifs, which we can just assume that all of them are constant C, and there are 1 if and else outside the loop, they are also considered as constant, therefore, running time of the algorithm is C\*N+C and the Theta Notation is O(N) because the running time grows as N gets larger.
2. This algorithm won’t detect all problems because of it only check the sum of it and not the real integer, for example, 12 could be 3+9 and it also could be 4+8, thus if we alter the number in array A as long as the sum is still the same, it will pass the test. It will also pass the test if we just swap the number; 3+9 to 9+3, the sum will be the same, but it is a different array from array A.

Task 4

1. Function R1(key,a,b,A,B,N):

For 0 <= i < N:

If B[(a \* A[i] + b) mod N] = A[i]

If A[i] = key:

Return i

Else:

Return -2

Return -1

1. A method that we could do to adapt repeated integers in the array A is by using linked list method. We can stack the integers with equal value so that it will not disrupt the order in array B. To implement this in coding, I personally will use 2D array to store multiple same values. Another method can be used, and it is called linear probe. The way that this works is but putting the value next to the index that we supposed to put it if we already have an element residing in that index. Both have their pros and cons.

Task 5:

The way that I store this data is by altering the index by 1, it is like the Caesar Cypher method of encrypting. A[i] will be stored in B[i+1], and if B reaches the end of its array, it will store it on the first value instead. The value A[i] will be multiplied by a secret number which the user can edit. The way this algorithm checks for hardware failure is by checking each and every one of the indexes. It will check whether all value A is present in B with the rule applied. The search itself will be a linear search, but it could return multiple value as the value in array A could have duplicates.

Key = what we want to find

A = data that we want to store

B = place to store A

N = length of array A

Secret = secret number for storing A in B

1. Function R3(key,A,B,N,secret):

Matched\_key = empty array

For 0 <= I < N:

If i != N-1:

if B[i+1]/secret != A[i]:

return -2

else:

if B[0]/secret != A[i]:

return -2

if B[i]/secret = key:

if i - 1 = -1:

add N-1 to matched\_key

else:

add i-1 to matched\_key

if length of matched\_key != 0:

return matched\_key

else:

return -1

1. This pseudocode works because the first if else condition will check whether the array B is filled with element from array A that has been altered by 1 to the right and multiply by the secret number. If there is one value that filled the requirement of error, it will return -2 immediately. If the checking does not face any problem, it will go through the linear search. If any of the value in B matched the key, it will append or add the matched indexed to array matched\_key, this index is based on A. When the searching completes, we have to see whether there is a value in matched\_key, if there is, we will return the value, if there is not a single value inside the array, we will return -1 instead. B could be manually inserted or returned from createB algorithm

Function create(A,N,secret):

B = [-1]\*N

for i in range(N):

if i == N -1:

B[0] = A[i]\*secret

else:

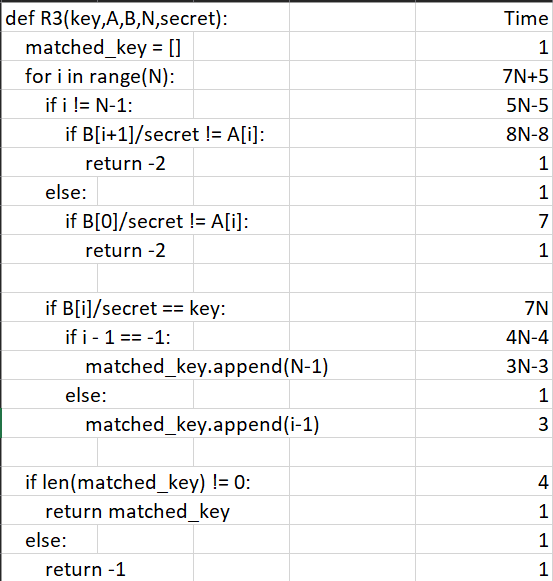
B[i+1] = A[i]\*secret

return B

This algorithm works by altering index of array A to the right by 1. Every iteration will go through A and put that value into B, this is one way that we can get B with minimum error.

1. The best-case of this algorithm is when there exist and error with second element of array B (B[1]). This is because when we access the i = 0, we check the index 1 instead, because i+1 condition.

The worst-case of this algorithm is when all of the element inside the array B is the key and there is no error inside array B. For example, A = [1,1,1,1,1,1], and the key is 3 so B = [3,3,3,3,3,3]. The second worst-case would be when the key cannot be found, as it will go through all block of code.



The running time of the best-case would be T(N) = O(1), the running time of the worst-case would be 34N + 3. The minus comes from the fact that each N-1 iteration would go to if and at least 1 will go to the else in the worst-case, it’s the same as the second if else pair. The rest of them will go through N times with some constant to it. In the end, we take the larger pair which is when the length of matched\_key is not 0. Therefore, our final running time for the worst-case is 34N + 3

1. To find the theta notion, we have to find the tightest bound in the middle of lower and upper bound. We can say that our fastest running time is our lower bound and the worst running time is our upper bound:

1 <= theta <= 34N+3

It is safe to say that our theta notation is O(N), because it fulfils the requirement and because N is dominant in this expression. Therefore, the theta notation for this algorithm is O(N), which grows linearly to N