**COMP 3270 Homework 1**

100 points. Due by **11:59pm (midnight) on Wednesday, June 7th, 2023**

Instructions:

1. Submissions not handed on the due date and time **will not** be accepted unless prior permission has been granted or there is a valid and verifiable excuse.
2. Think carefully; formulate your answers, and then write them out concisely using English, logic, mathematics and pseudocode (no programming language syntax).
3. Type your final answers in this Word document and submit online through Canvas.
4. Don’t turn in handwritten answers with scribbling, cross-outs, erasures, etc. If an answer is unreadable, it will earn zero points. **Neatly and cleanly handwritten submissions are also acceptable**.

1. (10 points) Using the first strategy in the Simplified Boggle problem we’ve created strings from the board and looked these up in the dictionary (see slide 21 in L3-Computational Problem Solving). Instead, **what if you go in the other direction**: get each **full** word **from the dictionary** and **see if it is on the board**?

* Develop this strategy further and write down the corresponding algorithm on paper. Assume the dictionary contains **m words** of **length at most n characters** and **the only function you have** for accessing it is **get\_next\_word** that will fetch the next word (or return nil when all words have been fetched). (5 points)
* Which strategy/algorithm is most efficient? Why? (3 points)
* Can this strategy be made more efficient if the dictionary is organized in a particular way and/or provided other functions? Which functions? How much more efficient will this strategy become? (2 points)

**Step 1: Algorithm for Strategy #2**

1. Initialize an empty set called "board\_words" to store the words found on the board.
2. Call the get\_next\_word function to fetch the first word from the dictionary.
3. While the fetched word is not nil:
   * Iterate over each cell on the board:
     + Start from the current cell and explore all possible directions (up, down, left, right, diagonal) to form the fetched word.
     + If a match is found, add the word to the board\_words set.

4. Return the board\_words set.

**Explanation:**

In this strategy, we iterate over each word in the dictionary and check if it can be formed on the board by exploring all possible directions from each cell. By doing so, we eliminate the need to generate all possible strings from the board, reducing the search space.

**Step 2: Efficiency of Strategy #2**

Strategy #2 is generally more efficient than Strategy #1. Here's why:

1. Reduced search space: Strategy #2 directly searches for valid words on the board instead of generating all possible strings. This reduces the search space and computational effort, especially when the board is large or the dictionary contains a lot of words.
2. Early termination: Strategy #2 allows for early termination when a word is not found on the board. If a partial word doesn't match any prefix on the board, there's no need to continue exploring that path. This can save significant computational time.
3. Flexibility: Strategy #2 can be more flexible when it comes to variations in board sizes or dictionary lengths. It doesn't rely on generating and comparing all possible strings, making it adaptable to different scenarios.

**Step 3: Optimizing Strategy #2**

To make Strategy #2 more efficient, we can consider organizing the dictionary in a particular way or providing additional functions:

1. Trie data structure: Organizing the dictionary as a trie can improve the efficiency of the search process. Each node in the trie represents a prefix or a complete word. By traversing the trie according to the characters on the board, we can quickly determine if a word exists.
2. Pruning techniques: Implementing pruning techniques can further optimize the search process. For example, we can prune branches of the trie that don't have a matching prefix on the board, reducing unnecessary exploration.
3. Length-based indexing: If the dictionary is sorted by word length, we can optimize the search by focusing on words of appropriate lengths first. This approach allows us to avoid unnecessary checks for longer words that cannot be formed on the board due to size limitations.

By combining these optimizations, the efficiency of Strategy #2 can be significantly improved, reducing the search time and computational resources required to find valid words on the board. However, the exact amount of improvement would depend on factors such as the size of the board, the length of the words, and the distribution of words in the dictionary.

2. (5 points) *Computational problem solving: Developing strategies:* Given a string, *S*, of *n* digits in the range from 0 to 9, describe an efficient strategy for converting *S* into the integer it represents.

Calculation to change string over to integer:

1. Sring of length n is put away in factor s.
2. Traverse through string s of length n by perusing character by character.

Step 1: convert the character to whole number by transformation work.

Step 2: display the whole number.

Step 3: end

Running time: traversing through the stron of length n, utilizing a circle, circle differs from 0 to n-1. So the running reason of the calculation if o(n).

3. (5 points) *Computational problem solving: Estimating problem solving time:* Suppose there are three algorithms to solve a problem- a O(n) algorithm (A1), a O(nlogn) algorithm (A2) and a O(n2) algorithm (A3) where log is to the base 2. Using the techniques and assumptions in slide set L2-Buffet(SelectionProblem).ppt, determine how long in seconds it will take for each algorithm to solve a problem of size 200 million. You must show your work to get credit, i.e., a correct answer without showing how it was arrived at will receive zero credit.

there is a computer running at 4 x 109 clock cycles per second. This machine typically requires about 200 clock cycles to execute one computation step.

So, the computer will execute ( 4x 109)/(200) = 2 x 107 ops per second.

Hence,

A1 O(n) will take : (200 x 106) / ( 2 x 107 ) = 10 seconds

A2 O(n log(n) ) will take = [ 2 x 108 x log2( 2 x 108 ) ] / [ 2 x 107 ] = 275.75 seconds

A3 O(n2) will take = [ 2 x 108 ]2 / [ 2 x 107 ] = 2 x 1016-7 = 2 x 109 seconds = 64.300411 years ( approx.)

4. (6 points) *Computational problem solving: Developing strategies*

Explain a correct and efficient **strategy** to check what the maximum difference is between any pair of numbers in an array containing n numbers. Your description should be such that the strategy is clear, but at the same time, the description should be at the level of a strategy, not an algorithm. Then state the total number of number pairs any algorithm using the strategy “compute the difference between every number pair in the array and select that pair with the largest difference” will have to consider as a function of n.

A correct and efficient strategy to check what the maximum difference is between any pair of numbers in an array containing n numbers would be to compute the difference between every number pair in the array and select that pair with the largest difference. The total number of number pairs any algorithm using the strategy "compute the difference between every number pair in the array and select that pair with the largest difference" will have to consider as a function of n is n(n-1)/2.

A more efficient strategy would be to sort the array and then compute the difference between the first and last numbers, the second and second-to-last numbers, the third and third-to-last numbers, etc. The total number of number pairs any algorithm using this strategy will have to consider as a function of n is n/2.

A even more efficient strategy would be to keep track of the minimum and maximum numbers seen so far as the array is being traversed. The total number of number pairs any algorithm using this strategy will have to consider as a function of n is 2n.

5. (10 points) *Computational problem solving: Understanding an algorithm and its strategy*

**Algorithm** Mystery(A[1..n])

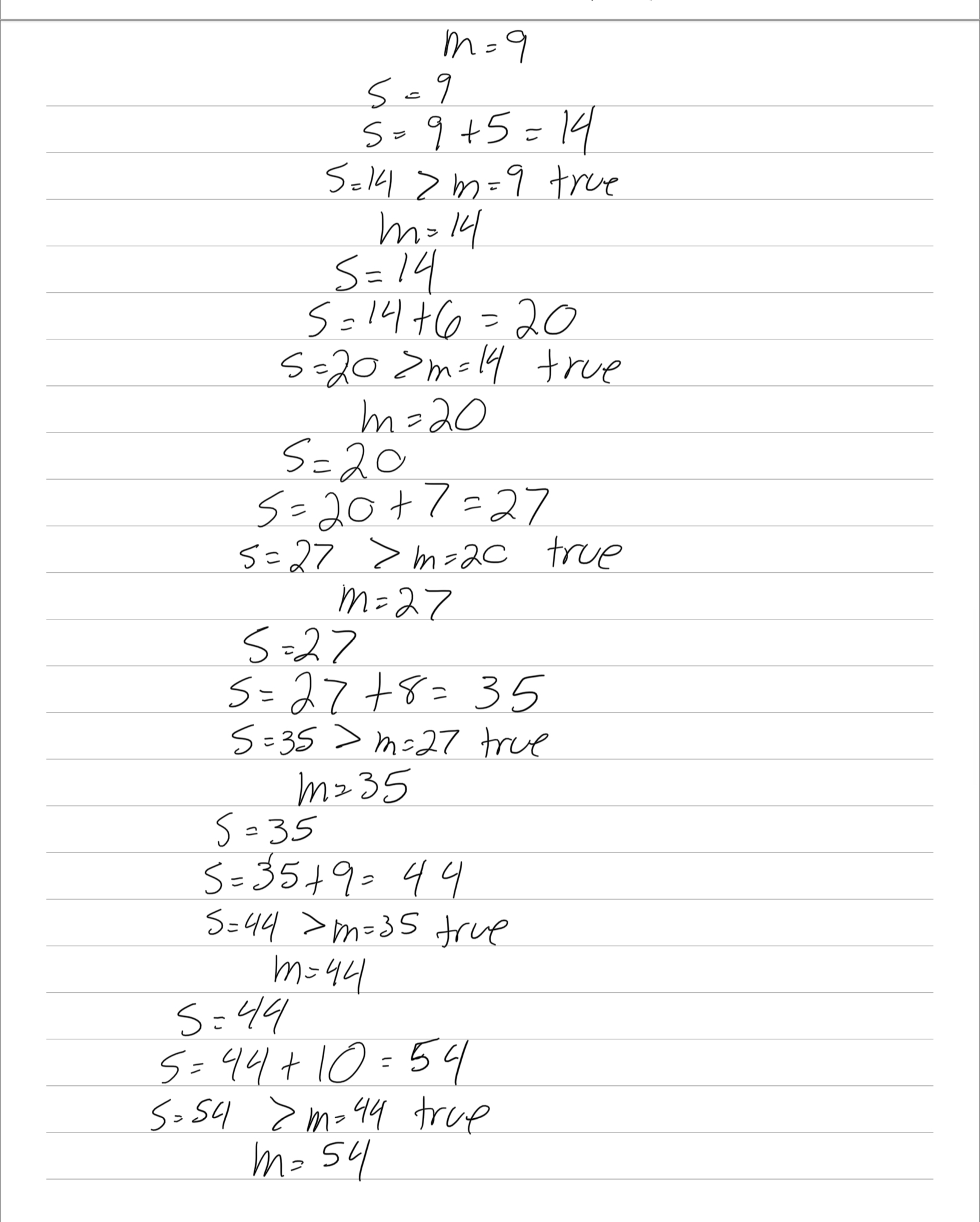
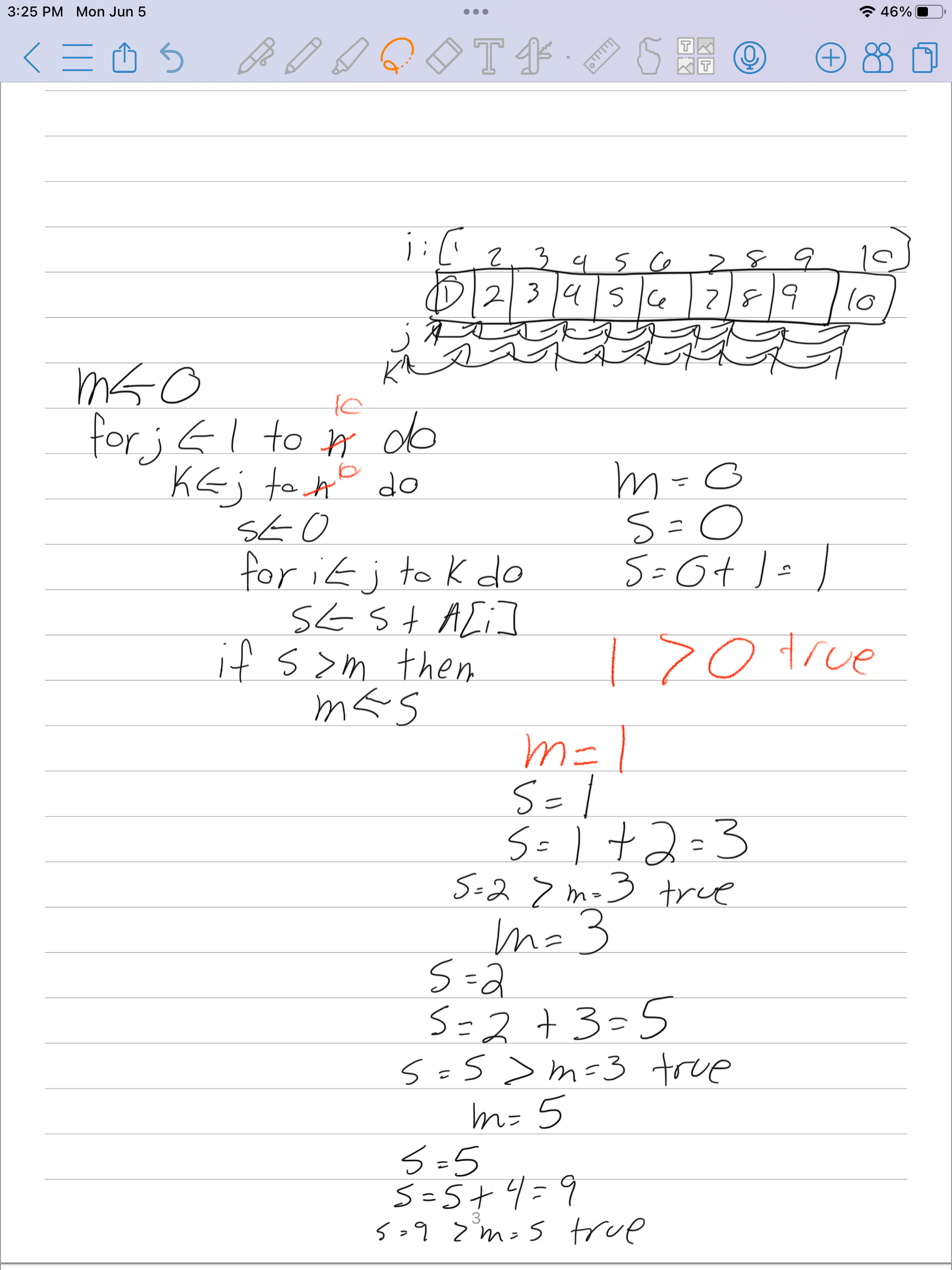
**Input**: An n-element array. Indexed from 1 to n

Text

Description automatically generated

1. Explain what the following algorithm outputs and simulate its operation on a valid input instance (e.g., an array of n elements - you can choose n to be 10)

For an input of 10 elements m = 54.



1. What is the approximate time complexity (running time) of the above algorithm (you can use Big-Oh notation) O(n^3) because you have O(n) with two nestled O(n) which makes it O(N^3)
2. How does the following algorithm improve the time complexity of the algorithm (what is its strategy)? What is its time complexity? we are basically creating a sum array to store our sums. We skip iterating through the array A[] twice and end up in O(n^2) time complexity.

Text

Description automatically generated

6. (9 points) *Computational problem solving: Calculating approximate complexity:*

Using the approach described in class (L5-Complexity.pptx), calculate the approximate complexity of Mystery algorithm above by filling in the table below.

|  |  |
| --- | --- |
| Step | Big-Oh complexity |
| 1 | O(1) |
| 2 | O(1) |
| 3 | O(n) |
| 4 | O(1) |
| 5 | O(n) |
| 6 | O(1) |
| 7 | O(1) |
| 8 | O(1) |
| 9 | O(1) |
| Complexity of the algorithm | O(n^2) |

7. (9 points) Calculate the detailed complexity T(n) of Mystery. Fill in the table below, then determine the expression for T(n) and simplify it to produce a polynomial in n.

|  |  |  |
| --- | --- | --- |
| Step | Cost of each execution | Total # of times executed |
| 1 | 1 | n |
| 2 | 1 | N+1 |
| 3 | 1 | N^2 |
| 4 | 1 | N^3 |
| 5 | 1 | N^3 |
| 6 | 1 | N^3 |
| 7 | 1 | N^3 |
| 8 | 1 | N^2 |
| 9 | 1 | n |

T(n) = 4n^3 + 2n^2 + 3n

The total number of times each step is executed depends on the values of j and k, which are both iterating from 1 to n. Step 1 and Step 2 are executed n times each. Step 3 is executed n^2 times because it is nested inside the loop for j, and Step 4, 5, 6, and 7 are also executed n^3 times because they are nested inside the loop for k. Step 8 and Step 9 are executed n^2 and n times respectively.

To calculate the detailed complexity T(n), we add up the total number of times each step is executed and multiply it by the cost of each execution:

T(n) = 1 \* n + 1 \* n + 1 \* n^2 + 1 \* n^3 + 1 \* n^3 + 1 \* n^3 + 1 \* n^3 + 1 \* n^2 + 1 \* n

Simplifying the expression, we get:

T(n) = n + n + n^2 + n^3 + n^3 + n^3 + n^3 + n^2 + n

T(n) = n^3 + n^3 + n^3 + n^3 + n^2 + n^2 + n + n + n

T(n) = 4n^3 + 2n^2 + 3n

So the polynomial expression for T(n) is 4n^3 + 2n^2 + 3n, which represents the detailed complexity of Mystery for the given input size n.

8. (10 points) *Computational problem solving: Proving correctness:* Complete the proof by contradiction this algorithm to compute the Fibonacci numbers is correct.

function fib(n)

1. if n=0 then return(1)

2. if n=1 then return(1)

3. last=1

4. current=1

5. for i=2 to n do

6. temp=last+current

7. last=current

8. current=temp

9. return(current)

1. Assume the algorithm is incorrect.
2. Fibonacci numbers are defined as F0=1, F1=1, Fi=Fi-1+Fi-2 for i>1.
3. So the assumption in (1) implies that there is at least one input parameter n=k, k≥0, for which the algorithm will produce an incorrect answer.
4. **If k=0, then if statement in the step 1 will be executed and returns 0th Fibonacci number that is ‘1’.**

**If k=1, if statement in the step 2 will be executed and returns 0th Fibonacci number that is ‘1’.**

So in both cases the algorithm returns the correct answer.

1. This implies that there has to be at least one integer k>1, so that when n=k the algorithm does not return the correct answer Fk=Fk-1+Fk-2.
2. When n=k and k>1 **the first two steps will be skipped**, and steps 3-9 will be executed.
3. If k=2, the for loop in steps 5-8 will be executed exactly once. By step 6, temp = last + current = 1 + 1 = F0 + F1. Then step 7 updates last to be equal to current = F1. Step 7 updates current to be equal to temp which is F0 + F1. So the value returned in step 9 is current = F0 + F1 = F2. This is the correct answer. So the k for which the algorithm fails must be greater than 2.
4. If k=3, **the for loop in steps 5-8 will be executed exactly twice. In the first iteration of loop, By step6, temp=last +current= 1+1 =F0+F1. Then step 7 updates last to be equal to current=F1. Step 8 updates current to be equal to temp which is F0+F1.**

**Before moving to second iteration last= F1=1 and current=F2=2.**

**By step 6, temp=last + current=F1+F2=1+2=3. Step 7 updates last to be equal to current=F2. Step 8 updates current to be equal to temp which is F1+F2. So the value returned in step 9 is current=F1+F2=F3. This is the correct answer. So the k for which the algorithm fails must be greater than 3.**

1. But if k= 4, **the for loop in steps 5-8 will be executed exactly thrice.**

**In the first iteration of loop, By step6, temp=last +current= 1+1 =F0+F1. Then step 7 updates last to be equal to current=F1. Step 8 updates current to be equal to temp which is F0+F1.**

**Before moving to second iteration last= F1=1 and current=F2=2.**

**By step 6, temp=last + current=F1+F2=1+2=3. Step 7 updates last to be equal to current=F2. Step 8 updates current to be equal to temp which is F1+F2.**

**Before moving to third iteration last= F2=2 and current=F3=3.**

**By step 6, temp=last + current=F2+F3=2+3=5. Step 7 updates last to be equal to current=F3. Step 8 updates current to be equal to temp which is F2+F3.**

**So the value returned in step 9 is current= F2+F3=F4. This is the correct answer. So the k for which the algorithm fails must be greater than 4.**

1. The above argument can be repeated to show that **the algorithm returns correct answer.**
2. That is, for all k > 1 the algorithm returns the correct k-th Fibonacci number.
3. So there is no k for which the algorithm will return a value not equal to Fk-1+Fk-2. This contradicts (3).
4. Therefore, the algorithm must be correct.

9. (a) (6 points) *Computational problem solving: Algorithm design:* Describe a recursive algorithm to reverse a string that uses the strategy of swapping the first and last characters and recursively reversing the rest of the string. Assume the string is passed to the algorithm as an array A of characters, A[p…q], where the array has starting index p and ending index q, and the length of the string is n=q–p+1. The algorithm should have only one base case, when it gets an empty string. Assume you have a swap(A[i],A[j]) function available that will swap the characters in cells i and j. Write the algorithm using pseudocode without any programming language specific syntax. Your algorithm should be correct as per the technical definition of correctness.

Algorithm:

Recursive – revserse (A[p, --q)

{

If (p <= q)

{

Swap (A[p], A[q]); //method to swap

//elements of position pq is A

//recursive call

Recursive—reverse(A[p+1...q-1]

}

}

(b) (8 points) Draw your algorithm’s recursion tree on input string “i<33270!”- remember to show inputs and outputs of each recursive execution including the execution of any base cases.

Input string: I <33270!

String length: n = 8

P = 0 q = 7. Let recursive-reverse = R –r

|  |  |
| --- | --- |
| algorith | string |
| 1. R-r(A[0..7] | I < 33270! |
| 1. Swap(A[0], A[7]) | ! < 33270i |
| 1. R-r[A[1...6]] | ! < 33270i |
| 1. Swap(A[1], A[6]) | !03327<i |
| 1. R-r[A[2..5]] | !03327<i |
| 1. Swap[A[2], A[5]] | !07334<i |
| 1. R-r[A[3..4] | !07323<i |
| 1. Swap{A[3], A[4]] | !07233<i |
| 1. R-r [A[4..3]] | Final revsed |
| Break here | String: !07233<i |
| Because p > q |  |

10. (10 points) *Computational problem solving: Proving correctness:*

Function g (n: nonnegative integer)

if n ≤ 1 then return(n)

else return(5\*g(n-1) – 6\*g(n-2))

Prove by induction that algorithm g is correct, if it is intended to compute the function 3n-2n for all n ≥ 0.

Base Case Proof:

N = 1 for the function it would return 1. 3^n – 2^n = 3^1 – 2^1 = 3-2 = 1.

Base case is n = 1.

Inductive Hypothesis:

Let us assume that for n <= k, the function g(k) returns 3k - 2k and g(k-1) = 3k-1 - 2k-1

Inductive Step:

Now, we would need to show that for n = k + 1, the function g(k + 1) returns 3k+1 - 2k+1.

**Proof:**

g(k + 1) = 5\*g(k) - 6\*g(k-1)

= 5 \* (3k - 2k) - 6(3k-1 - 2k-1)

= 15\*3k-1 - 10\*2k-1 - 6\*3k-1 + 6\*2k-1

= 9\*3k-1 - 4\*2k-1

= 3k+1 - 2k+1 (proved)

12. (12 points) *Computational problem solving: Proving correctness:* The algorithm of Q.8 can also be proven correct using the Loop Invariant method. The proof will first show that it will correctly compute F0 & F1 by virtue of lines 1 and 2, and then show that it will correctly compute Fn, n>1, using the LI technique on the for loop. For this latter part of the correctness proof, complete the Loop Invariant below by filing in the blanks. Then complete the three parts of the rest of the proof.

function fib(n)

1. if n=0 then return(1)

2. if n=1 then return(1)

3. last=1

4. current=1

5. for i=2 to n do

6. temp=last+current

7. last=current

8. current=temp

9. return(current)

Loop Invariant:

Before any execution of the for loop of line 5 in which the loop variable i=k, 2≤k≤n, the variable last will contain (k-2)th Fibonacci number F(k-2) and the variable current will contain (k-1)th Fibonacci number F(k-1.

Initialization:

Before the first iteration, with i=2, we have **current** containing F(1)=1 and **last** containing F(0)=1. Hence, the loop invariant is true.

Maintenance:

Let before the iteration of the for loop with i=k, the variable **current** contains F(k-1) and **last** contains F(k-2). Then, after the iteration (before the iteration with i=k+1), *temp=last+current* sets **temp** to be F(k-1)+F(k-2)=F(k), *last=current* sets **last** as F(k-1) and *current=temp* sets **current** as F(k). So, the loop invariant remains true.

Termination:

The for loop runs till i=n, so the final value of **current** when i=n is F(n). Since this value is returned, the function successfully computes the nth Fibonacci number.