HW1

1.

(a)To develop a strategy

Initialize an empty list called "found\_words" to store the found words.

Initialize a variable “word” and set it to the result of the function get\_next\_word() from the dictionary.

For each word Initialize a variable called "word\_found" as True, Iterate through each character, "char," in the word.

For each character, Initialize a variable called "char\_found" as False, Iterate through each row, "row," in the board.

For each row, iterate through each column, "column," in the board.

If the character at the current position (row, column) on the board matches the current character, "char," in the word, set "char\_found" as True, and break out of the inner loops.

If "char\_found" is False, set "word\_found" as False, break out of the outer loop, and continue to the next word in the dictionary.

If "word\_found" is True, append the word to the "found\_words" list.

Return the "found\_words" list containing all the words found on the board.

(B) The first Strategy is more efficient because It performs dictionary lookups for each substrings, and it allow early termination when substring is not valid word, and It reduce the search space.

(C) yes, it can be more efficient if the dictionary is organize in a tire data structure, since the trie reduce the search space and improve the process.

(2) First, a string should be stored as a variable. Then traverse the string by reading each individual character, a conversion function should be used to convert characters to integers, the last step should display the integer, by using a loop, we traverse the string and our running time should be 𝑂(n)

(3) there is a computer running at 4 x 109 clock cycles per second. The machine requires about 200 clock cycles to execute one computation step.

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

Algorithm 1: O(n) : (200 x 106) / ( 2 x 107 ) = 10 seconds

Algorithm 2: O(n log(n)): [ 2 x 108 x log2 ( 2 x 108 ) ] / [ 2 x 107 ] = 275.75 seconds

Algorithm 3: O(n2): [ 2 x 108 ]2 / [ 2 x 107 ] = 2 x 1016-7 = 2 x 109seconds

(4) declare and initialize variables in the array to minimum and maximum, and maximum difference, my loop would run from 1 to array length -1.

Each loop should check if the variable is less or greater than the minimum value.

If it is less, then variable will be marked as a minimum element, otherwise then recursively call the maximum to find maximum element, the loop will find the difference of the minimum and maximum variable.

(5) this program calculates the maximum subarray sum of an input array A. We are iterating through each element of A[] and from that element's position we are calculating subarray's sum till the end of the array. In the end we are just returning the maximum sum which is store in m. All the partial sums are stored in s.

let n = 10.

input: A = [1, -2 , 3, 4, -5, 6, -7, 8, 9, -10];

We are going to get the following sums:

K =1  
s = 1

m = 1  
k = 1  
s = -2  
m = 1  
…. until k = 10, s = -10, m = 9, the algorithm output 9 is the maximum subarray sum for the array.

b) The Big-O notation comes to be O(n^3) = O(n\*n\*n)since we are using 3 nested loops.

C)in this algorithm we can see that the sum is computed at repeated times.

To improve the performance, we added the array before and stored in the array and we used that array for our convenience. Thus reduced the time complexity.

Time complexity:

O(n) To create the array so.

O(n\* (n+1)/2)

Total time complexity: O(n)+O(n(n+1)/2). Which is O(n^2) = O(n \* n)

(6)

|  |  |
| --- | --- |
| 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)

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

T(n) = 6n^2 + 9n + 3

(8)

4. If k is 0, then the if statement in step 1will be executed and returns the integer in the array at

place 0. Fibonacci number is "1".

If k is 1, then if statement in step 2 will run and returns the same value if k was 0, Fibonacci number is "1".

6. When n=k and k>1 the first two steps will be skipped

8. 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.

9. 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.

10. The above argument can be repeated to show that the algorithm returns the correct answer.

(9) (a)

reverse.recursive(A[p,...q])

If (p <=q)

Swap (A[p], A[q] )

reverse.recursive(A[p+1.......q-1]

(b)

String length n = 8, p = 0, q = 7

1.Recursive\_reverse [A[0,...7] = i<33270!

2.Swap (A[0], A[7]) =! <33270i

3. Recursive\_reverse(A[1,...6]) =!<33270i

4.Swap (A[1], A[6]) = !03327<i

5. Recursive\_reverse(A[2,...5]) = !03327<i

6.swap(A[2], A[5]) = !07323<i

7. Recursive\_reverse(A[3,...4]) = !07323<i

8.Swap (A[3], A[4]) = !07233<i

(10) Base cases: when n=0, n=1

for n = 0, g(0)= 3^0-2^0 = 0 and the algorithm returns to 0.

for n = 1, g(1) = 3^1-2^1 = 1 and the algorithm returns to 1 and the base case hold.

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.

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

(12)

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 2 and the variable current will contain 3

Initialization: fib(k) = fib(k-1) + fib(k-2) when k>1.

Maintenance: Current is updated with the function fib(k-1) + fib(k-2) for every value k>1.

Termination:

Loop ends for k, and current will equal the sum of the two previous values.