CSE222 HW4

Time Complexity Analysis

Q1

Base case for this function is being traversed all the text that is given subtext is searched in. And in every recursive call one character is traversed thus the worst case is $\theta(\mathrm{n})$ for text with n characters. Best case is when the searched text is successfully traversed, there may or may not be characters before searched text but traversing that part is constant time since it is not necessarily relative to n thus best case is $\theta(\mathrm{m}) + \theta(1) = \theta(\mathrm{m})$ when searched text is m characters long. Of course, these can be said because all the operations are constant time in the function (equality check and subtraction), and only one more call is done for each call at max, and occurrence value doesn't really matter since it can only affect as a multiplier and $\mathrm{c}\theta(\mathrm{m}) = \theta(\mathrm{m})$. Therefore, the function time complexity is linear in either case.

Q2

Base case for this function is not being able to find given integers, which can only be made sure after searching the array, and since it is searched like binary search it is $\theta(\log \mathrm{n})$. But the worst case is when array is filled only with given integers and since when given integers are found their neighbour value is found by iterating the elements one by one, therefore making it $\theta(\mathrm{n})$. Best case for the function is having given integers only at the ends of the array, where it's checked first making it $\theta(1)$. Therefore, time complexity of this function is O(n).

Q3

Base case for this function is having no elements to further iterate to in array, since it is only a return statement it is constant time. Then list of integer lists whose elements' sum with current element is equal to given integer gotten through first recursive call, there may or may not be any list satisfying this. Though there can only be a maximum of $(\mathrm{n} - \mathrm{i})$ lists satisfying it since the list has to start from current element position (i) and can end at any one of the remaining $(\mathrm{n} - \mathrm{i})$ elements on the right side of the current element. Though there may be sub lists that satisfy the given sum that don't start at current element position so to check that head element position is also iterated through a second recursive call. Therefore, the time complexity of this function is $\theta(\mathrm{n}) * \theta(\mathrm{n}) = \theta(\mathrm{n}^2)$

Q4

The function returns the multiplication of given integers.

For one-digit integers the function returns their multiplication in the base case. From now on consider the variables in multiplication when only there's a multiplication operator (·), otherwise think them as integers and variables representing the digit numbers. Now suppose given integers are two-digit integers xy and ab such that sum of their integers within them is less than 10. Let's examine the function behaviour for these integers:

$\mathrm{int1} = \mathrm{x}, \mathrm{int2} = \mathrm{y}$

$\mathrm{int3} = \mathrm{a}, \mathrm{int4} = \mathrm{b}$

$\mathrm{sub0} = \mathrm{y} \cdot \mathrm{b}$

$\mathrm{sub1} = (\mathrm{x} + \mathrm{y}) \cdot (\mathrm{a} + \mathrm{b}) = \mathrm{x} \cdot \mathrm{a} + \mathrm{y} \cdot \mathrm{a} + \mathrm{x} \cdot \mathrm{b} + \mathrm{y} \cdot \mathrm{b}$

$\mathrm{sub2} = \mathrm{x} \cdot \mathrm{a}$

$\mathrm{sub2} \cdot 10^{(2\cdot 1)} = \mathrm{x} \cdot \mathrm{a} \cdot 100$

$(\mathrm{sub1} - \mathrm{sub2} - \mathrm{sub0}) \cdot 10^1 = \mathrm{y} \cdot \mathrm{a} \cdot 10 + \mathrm{x} \cdot \mathrm{b} \cdot 10$

thus return value is

$= \mathrm{x} \cdot \mathrm{a} \cdot 100 + \mathrm{y} \cdot \mathrm{a} \cdot 10 + \mathrm{x} \cdot \mathrm{b} \cdot 10 + \mathrm{y} \cdot \mathrm{b}$

$= (\mathrm{x} \cdot 10 + \mathrm{y}) \cdot \mathrm{a} \cdot 10 + (\mathrm{x} \cdot 10 + \mathrm{y}) \cdot \mathrm{b}$

$= (\mathrm{x} \cdot 10 + \mathrm{y}) \cdot (\mathrm{a} \cdot 10 + \mathrm{b}) = (\mathrm{xy}) \cdot (\mathrm{ab}) = \mathrm{xy} \cdot \mathrm{ab}$

It is seen that for xy and ab function returns their multiplication. If the sum of digits within the integers were greater than or equal to 10 the sum of digits of that sum is guaranteed to be less than 10 and since we just proved for such integers that their multiplication is returned the sub1 would still be equal to $(\mathrm{x} + \mathrm{y}) \cdot (\mathrm{a} + \mathrm{b})$. Therefore, it is seen that function returns the multiplication of every integer less than 100.

Since function is recursive it will split given integers until they are several two-digit or one-digit integers, thus the function will return the multiplication result of any given integers.

For the time complexity let $M(n)$ give the maximum number of multiplications that can be done in base case for n digits numbers. We know that function calls itself for right halves of the numbers then sum of their halves and then for the left halves. So

$$\mathrm{M}(n) = \mathrm{M}\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + \mathrm{M}\left(1 + max\left(\left\lfloor\frac{n}{2}\right\rfloor, n - \left\lfloor\frac{n}{2}\right\rfloor\right)\right) + \mathrm{M}\left(n - \left\lfloor\frac{n}{2}\right\rfloor\right), \text{when } n > 3$$
$$M(1) = 1, M(2) = 5, M(3) = 17$$

If we assume that n is even, then equation becomes

$$M(n) = 2M\left(\frac{n}{2}\right) + M\left(1 + \frac{n}{2}\right)$$
$$\text{since } M\left(1 + \frac{n}{2}\right) > M\left(\frac{n}{2}\right)$$
$$M(n) > 3M\left(\frac{n}{2}\right) = 3^2 M\left(\frac{n}{2^2}\right) = 3^k M\left(\frac{n}{2^k}\right), \text{let } k = log_2 n$$
$$M(n) > 3^{log_2 n} M(1) = 3^{log_2 n} = n^{log_2 3}$$

And if n was odd result would be even greater thus $M(n) = \Omega\left(n^{log_2 3}\right)$. I couldn't come up with a proof, but I think that it is also $\theta\left(n^{log_2 3}\right)$, since I found a good approximation $A(n) = \sqrt{23}x^{log_2 3}$ with error percentage less than 5% for n<1,000,000.

Inductive Proofs

Q1

The base case is when there's no remaining text to search in so given text couldn't been found therefore it's not in it, so -1 is returned. And if the given text to be searched is fully traversed, the index of its checked character being equal to its length, then it means every character is matched and therefore the text is found, and if occurrence parameter value is 1 it means this match should be returned, so i-j is returned since i points to the end of the searched text in the text that is searched in and j is the length of the searched text.

It is seen that base case and best case are working correctly. Now assume that for every occurrence value less than k the function either finds the text and returns its index. For occurrence = k, it finds the first match and calls itself as occurrence = k-1 for the rest of the string, and since function works for occurrence values less than k this also works. If the first match was not found this is handled by the base case and occurrence value doesn't matter.

We know functions works correctly when there's no remaining text to search or text to search in. Assume it also works for texts with less than n characters to search in and texts less than m characters to search. Now when n characters long text is searched for m characters long text it checks if first characters are same or not, if so it calls itself for m-1 characters long searched text and if not it calls itself again with m characters long searched text, and in either case the length of text that will be searched in is n-1. Therefore, it also works for n characters long text to search in and m character long text to search.

Q2

Base case for this function is not finding given integers which then -1 is returned, this is correct behaviour. Now assume that function works for subarrays of length less than n for values *floor* and *ceil*. For n element long subarray the function checks if first element is greater than floor if it is than first element of new subarray is half n elements before the current's. If it is less then it is half n elements after. If they are same function is called for the subarray with next element as the first element, making it n-1 long. For ceil the process is same but reversed. Now it is seen that there are cases that the next subarray to be checked gets bigger, but the function is called with given array when it's first called, and subarray to be checked in first recursive call happens to bigger than original array this is just handled in base case, and if this does not happen then the subarrays that are checked always be less than the original array with length n. Therefore, since function works for subarrays with length of less than n, it also works for arrays with length n.