DAA Assignment Report

1. *Introduction*:

The assignment deals with numbers called intals. These are non-negative integers that have arbitrary length but for assignment purpose 1000 is its maximum limit. In a language like C, there are no data structures to store such a long integer. The normal data structure like int, long int, long long int, etc have their own maximum limits but the length of intal is much greater than those limits. The int data structure can accept numbers in the range -2,147,483,648 to 2,147,483,647 and so on. But our intals have a maximum of 1000 decimal digits. Hence, the string data structure is used to store such intals which can accept up to 1000 or even more than those characters.

These intals form a special class of numbers which are integers hence should follow all general operations of integers. But they are stored in a string and performing arithmetic operations require a different logic. This is a challenge that is to be submitted in the given assignment. String, as usual, is represented as an array of characters in C hence directly performing operations is not possible. Traversing and performing the desired operation on each character is required.

Intals have huge applications in many fields. They are used in public key encryptions with RSA algorithms, computing the probabilities of certain real events, finding factorial for permutations and combinations, and other industrial applications. Hence their efficient execution in polynomial time is required. In this assignment, we develop a library of intal operations that run efficiently and can be imported in the above applications whenever required.

2. *Approach:*

This part deals with the logic followed to implement a few basic operations of intal and their asymptotic analysis. All the memory required to store the result of intals is dynamically allocated and freed once the work is done.

a) char * intal_add(char *intal1, char *intal2); - This function is used to add two intals and give the result in the form of an array of characters (string). This function allocates the max length of intal1 and intal2 to the result intal. It traverses both the intals and adds their respective characters keeping a track of carry. At the end when any one of the intal is fully traversed the remaining elements of another intal if present is added to carry and copied to result.
Time complexity:O(n)

Space complexity: O(1)
where n is the max length of intal1 and intal2.

b) int intal_compare(char *intal1, char *intal2); - It is used to compare the two intals and return 1 if intal1 > intal2, 0 if they are equal and -1 if intal1 < intal2. Based on lengths and strcmp they have been compared.
Time Complexity:O(n)
Space Complexity:O(1)
where n is the max length of intal1 and intal2.

c) char * intal_diff(char *intal1, char *intal2); - Difference of two intals is been calculated here. Initially, both are compared and the largest intal is found so that the difference is non-negative. Again the intals are traversed and the difference is calculated for individual corresponding characters keeping track of the borrow. Always the largest intal minus smallest intal is been done. The result is stored in a string that is dynamically allocated.
d) Space Complexity:O(1)
Time Complexity:O(n)
where n is the max length of intal1 and intal2.

e) char * intal_multiply(char *intal1, char *intal2); - The given function multiplies intal1 with intal2. The basic logic of multiplying each element of intal2 with intal1 and finally adding the result by shifting it is used here. Although there is a more efficient algorithm called Karatsuba algorithm its very complex to be implemented at this point. The result is stored in string of length strlen(intal1)+strlen(intal2)+1.
Time Complexity:O(n^2)
Space Complexity:O(1)
where n is the max length of intals.

f) char * intal_mod(char *intal1, char *intal2); - It gives the result of intal1 mod intal2. The normal process of repeated subtraction doesn't work as it exceeds the time limit. Hence only O(log n) number of subtractions are done by following the below procedure. Check if intal1 < intal2 if true then store it in final_intal and return it. Else multiply a copy of intal2 by 10 until it is greater than intal1. Once it is greater than intal1 just divide it by 10 and subtract intal2 from a copy of intal1 until it is less than intal2. Repeat the same procedure considering new intal1 into consideration and retaining old intal2. When the new value of intal1 is less than given intal2 just return it.

Time Complexity:O(log n) as approximately log n number of subtractions are done.
Space Complexity:O(n)
where n is the length of intal1.

g) char * intal_pow(char *intal1, unsigned int n); - This function calculates intal1^intal2. To find in log n time divide and conquer approach is used where a^n is divided as a^(n/2) and a^(n/2). Using recursion it is calculated in log n time. If n is odd then again a is multiplied to result.
Time Complexity: O(log n)
Space Complexity: O(log n) This is the stack space used in recursion.
Where n is the exponent.

h) char * intal_gcd(char *intal1, char *intal2); - It finds the Greatest Common Divisor(GCD) or Highest Common Factor(HCF). To efficiently compute GCD Euclid's Division Algorithm is used here. Intal2 is divided from intal1 to find quotient and remainder. In the next iteration, intal2 becomes intal1 and the remainder becomes intal2 until the remainder is zero. Finally, return intal2. Intal_mod is used here to find the remainder.
Time Complexity: O(n)
Space Complexity: O(n) because of the intal_mod function.
Where n = strlen(intal1), m = strlen(intal2)

i) char *intal_fibonacci(unsigned int n); - This function finds the nth Fibonacci number in series. The recurrence relation is used to find fibonacci number i.e., f(n) = f(n-1) + f(n-2). The intal_add function is used here. Finally the number is returned back in the form of string.
Time Complexity: O(n*m)
Space Complexity: O(1)
Where n - given argument m - time taken for intal_add function.

j) char *intal_factorial(unsigned int n); - It is used to find the factorial of number n. The intal_multiply is called iteratively to find n!. The final result is returned as a string.
Time Complexity: O(n*m)
Space Complexity: O(1)
Where n - given argument m - time taken for intal_multiply function.

k) char *intal_bincoeff(unsigned int n, unsigned int k); - The given function is used to find C(n,k) based on dynamic programming. The recurrence relation which is derived is C(n,k) = C(n-1,k) + C(n-1,k-1). Based on this a window of O(k) intals is allocated to find C(n,k). This is a part of the dynamic programming table. Once found is returned in the form of string. Further, another formula is also used to C(n,k) = C(n,n-k).
   Time Complexity: O(n*k)
   Space Complexity: O(k)
   Where n,k - given arguments.

l) int intal_min(char **arr, int n); - Minimum value is found out in array by iterating through it. The corresponding offset is returned as an integer.
   Time Complexity: O(n)
   Space Complexity: O(1)
   Where n - given argument.

m) int intal_max(char **arr, int n); - This function iterates through the whole array and returns the offset of the maximum value in the array.
   Time Complexity: O(n)
   Space Complexity: O(1)
   Where n - given argument.

n) int intal_search(char **arr, int n, const char *key); - It finds the first occurrence of the key in the array and returns it by iterating through the whole array.
   Time Complexity: O(n)
   Space Complexity: O(1)
   Where n - given argument.

o) int intal_binsearch(char **arr, int n, const char *key); - Used to find the first occurrence of an element in the sorted array by decrease and conquer where the array is divided into two halves and only one part is considered. The first occurrence of an element is found by including an extra condition whether the previous element is the same or not.
   Time Complexity: O(log n)
   Space Complexity: O(1)
   Where n - given argument.

p) void intal_sort(char **arr, int n); - This function is used to sort the array based on an efficient algorithm called heap sort. The array is assumed to be as a binary

tree and the largest value is brought to the top of the tree(max_heapify). It is an in-place and n*log n sorting algorithm.
Time Complexity: O(n*log n)
Space Complexity: O(1)
Where n - given argument.

q) char *coin_row_problem(char **arr, int n); - It is a dynamic programming problem which is solved by using a recurrence relation.

f(n) = max(f(n-2) + arr[n-1] ,f(n-1))

It is an optimization problem that maximizes the cost based on the given following constraints. A window of 3 intals is used to solve the current problem based on the above recurrence relation.
Time Complexity: O(n)
Space Complexity: O(1)
Where n - given argument

3. *Future task:*

We can extend this assignment by not restricting our intals to 1000 decimal digits but can have any arbitrary length. Further few complex operations could be added like using these in calculating binary digits which would help in encrypting data. If there were no restrictions then we can use many different data structures of C99 like int64_t, etc., to continue with the given task. Further many advanced facilities of C99 could be used. qsort(), bsearch() such inbuilt functions near to our applications could be used to develop more complex operations using intals.