RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSC209A
Course Name	Design and Analysis of Algorithms
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No	17ETCS002159
Semester/Year	04/2019
Course Leader/s	Pallavi R Kumar

:

Declaration Sheet				
Student Name	Satyajit Ghana			
Reg. No	17ETCS002159			
Programme	B.Tech		Semester/Year	04/2018
Course Code	CSC209A			
Course Title	Design and Analysis of Algorithms			
Course Date		to		
Course Leader	Pallavi R Kumar			

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

Figure No. **Title of the figure** **Pg.No.**

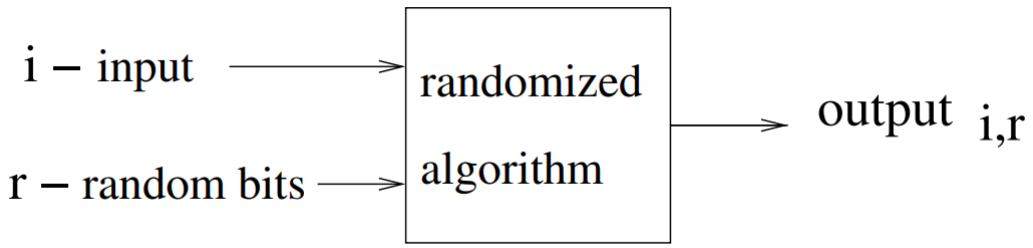
Question No. 1

Solution to Question No. 1 Part A:

A 1.1 Introduction:

A randomized algorithm A is an algorithm that at each new run receives, in addition to its input i, a new stream/string r of random bits which are then used to specify outcomes of the subsequent random choices (or coin tossing) during the execution of the algorithm.

Streams r of random bits are assumed to be independent of the input i for the algorithm A.



A randomized algorithm can be seen also in other ways:

- As an algorithm that may, from time to time, toss a coin, or read a (next) random bit from its special input stream of random bits, and then to proceed depending on the outcome of the coin tossing (or chosen random bit).
- As a nondeterministic-like algorithm which has a probability assigned to each possible transition.
- As a probability distribution on a set of deterministic algorithms -$\{A_i, p_i\}_{i=1}^{n}$

A 1.2 Advantages and drawbacks of randomized algorithm:

Advantages:

There are several important reasons why randomized algorithms are of increasing importance:

1. Randomized algorithms are often faster either from the worst-case asymptotic point of view or/and from the numerical implementations point of view;
2. Randomized algorithms are often (much) simpler than deterministic ones for the same problem;
3. Randomized algorithms are often easier to analyse and/or reason about especially when applied in counter-intuitive settings;
4. Randomized algorithms have often more easily interpretable outputs, which is of interests in applications where analyst's time rather than just computation time is of interest;

5. Randomized algorithms have been recently surprisingly successful when dealing with huge-data matrix computation problems.
6. Randomized numerical algorithms can often be organized better to exploit modern computer architectures. **(Prof Jozef Gruska, 2015)**

Drawbacks:

1. They are unrealistic.
2. Most of the tests are redundant.
3. More time is spent on analysing results.
4. The actual test results are random in the case of randomized software and random testing. Therefore, it is not possible to give an exact expected value.
5. One cannot recreate the test if data is not recorded which was used for testing.

A 1.3 Conclusion:

Randomized algorithms help in reducing the time and space complexity of programs, let's take a famous and widely used Monte Carlo Method,

Monte Carlo (MC) methods are a subset of computational algorithms that use the process of repeated random sampling to make numerical estimations of unknown parameters. They allow for the modeling of complex situations where many random variables are involved, and assessing the impact of risk. The uses of MC are incredibly wide-ranging, and have led to a number of groundbreaking discoveries in the fields of physics, game theory, and finance. There are a broad spectrum of Monte Carlo methods, but they all share the commonality that they rely on random number generation to solve deterministic problems.

Usage in High Energy Physics:

One major application of Monte Carlo that is near and dear to my heart is in the world of particle physics. In the quantum (very small-scale) world, things are not easily observable and this is especially true at the point of collision in a particle accelerator. MC methods allow physicists to run simulations of these events, based on the Standard Model, and parameters which have been determined from previous experiments. Huge scale projects like the LHC already produce an immense quantity of data, so N is already huge before we even start randomly sampling. So one small thing MC is useful for is probing the fundamental fabric of matter itself. **(Christopher Pease, 2018)**

Solution to Question No. 1 Part B:

B 1.1 Brute Force Algorithm for finding LCS:

We have two forms of brute force, the iterative method and the recursive method to solve this problem.

LCS-BRUTE-ITERATIVE(X, Y, m, n)

```
1. LCS = -inf
2. for k = 1 to m
3. length = 0;
4. for i = k to m
5.     for j = 1 to n
6.         if X[i] == Y[j]
7.             length = length+1
8.             i = i+1
9.             j = j+1
10. LCS = max(LCS, length)
11. return LCS
```

LCS-BRUTE-RECURSIVE(X, Y, m, n)

```
1. if m == -1 OR n == -1
2.     return 0
3. if X[m] == Y[n]
4.     return 1 + LCS-BRUTE-RESURSIVE(X, Y, m-1, n-1)
5. return max(LCS-BRUTE-RECURSIVE(X, Y, m-1, n), LCS-BRUTE-RECURSIVE(X,
   Y, m, n-1))
```

The recursive algorithm can be further optimized by memoization.

B 1.2 Dynamic programming-based algorithm for finding LCS:

LCS-DP(X, Y, m, n)

```
1. let L be a new (m+1)x(n+1) matrix
2. for i = 0 to m
3.     for j = 0 to n
4.         if i == 0 OR j == 0
5.             L[i][j] = 0
```

```
6.          else if X[i-1] == Y[j-1]
7.              L[i][j] = L[i-1][j-1] + 1;
8.          else
9.              L[i][j] = max(L[i-1][j], L[i][j-1])
10. return L[m][n]
```

We can view the algorithm above as just being a slightly smarter way of doing the original recursive algorithm, saving work by not repeating subproblem computations. But it can also be thought of as a way of computing the entries in the array L. The recursive algorithm controls what order we fill them in, but we'd get the same results if we filled them in in some other order. We might as well use something simpler, like a nested loop, that visits the array systematically. The only thing we have to worry about is that when we fill in a cell `L[i,j]`, we need to already know the values it depends on, namely in this case `L[i+1,j]`, `L[i,j+1]`, and `L[i+1,j+1]`. For this reason we'll traverse the array backwards, from the last row working up to the first and from the last column working up to the first.

(ICS 161, Lecture Notes, 1996)

B 1.3 Comparison of dynamic programming and Brute Force algorithms:

Brute-Force Algorithm:

The brute force algorithm runs for every character in the string and determines the longest substring for starting from that character, there are 3 nested loops here, the inner two loops determine the subsequences of maximum length, and the value of LCS is updated accordingly.

Since there are 2^m subsequence of X the time complexity is exponential, the space complexity is not relevant here since we aren't storing any values other than the LCS.

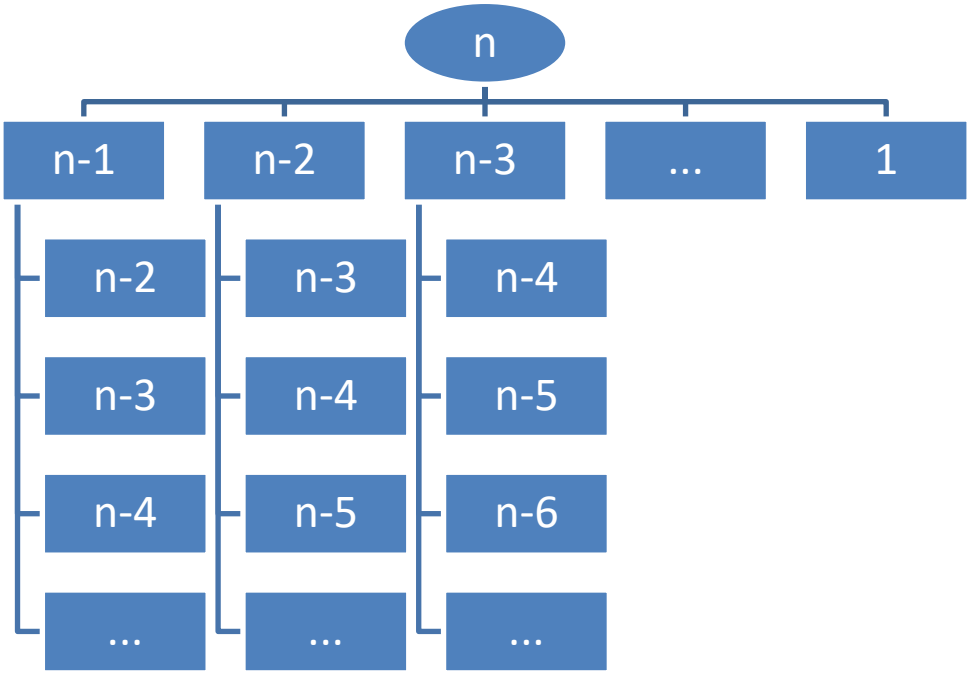
Checking = $O(m+n)$ time per subsequence

2^m subsequence of X

Worst-case running time = $O((m+n)2^m)$

The running time in this case is exponential and very inefficient.

Let's look at the Recursion Tree for the Brute Force Algorithm's Recursive Approach



As we can see that lots of nodes are repeated in the recursion tree

$$T(n) = T(n-1) + T(n-2)..+..T(1) + cn$$
$$T(1) = c$$
$$T(n) = O(n \cdot 2^n)$$

Dynamic Programming Algorithm:

The algorithm above in B1.2 has finished with the LCS length in X[0], Hirschberg finds the corresponding crossing place (m/2,k). He then solves recursively two LCS problems, one for A[0..m/2-1] and B[0..k-1] and one for A[m/2..m] and B[k..n]. The longest common subsequence is the concatenation of the sequences found by these two recursive calls.

It is not hard to see that this method uses linear space. What about time complexity? This is a recursive algorithm, with a time recurrence

$$T(m,n) = O(mn) + T\left(\frac{m}{2}, k\right) + T\left(\frac{m}{2}, n-k\right)$$

Think of this as sort of like quicksort -- we're breaking both strings into parts. But unlike quicksort it doesn't matter that the second string can be broken unequally. No matter what k is, the recurrence solves to O(mn). The easiest way to see this is to think about what it's doing in the array L. The main part of the algorithm visits the whole array, then the two calls visit two subarrays, one above and left of (m/2,k) and the other below and to the right. No matter what k is, the total size of these two subarrays is roughly mn/2. So instead we can write a simplified recurrence

$$T(mn) = O(mn) + T\left(\frac{mn}{2}\right)$$

which solves to O(mn) time total.

(ICS 161, Lecture Notes, 1996)

Let's take the example of the input strings ABCBDAB and BDCABA and make the DP Table for it.

			A	B	C	B	D	A	B
		-1	0	1	2	3	4	5	6
	-1	0	0	0	0	0	0	0	0
B	0	0	0	1	1	1	1	1	1
D	1	0	0	1	1	1	2	2	2
C	2	0	0	1	2	2	2	2	2
A	3	0	1	1	2	2	2	3	3
B	4	0	1	2	2	3	3	3	4
A	5	0	1	2	2	3	3	4	4

Figure 1 DP Table

The DP table is computed for the LCS, which is much more efficient than the brute force approach, which has a time complexity of $O((m+n)2^n)$.

B 1.4 Conclusion:

Dynamic Programming applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:

- Simple subproblems: the subproblems can be defined in terms of a few variables, such as j, k, l,
- m, and so on.
- Subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
- Subproblem overlap: the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up)

(M.T. Goodrich, R. Tamassia, 2015)

Dynamic programming is useful is your recursive algorithm finds itself reaching the same situations (input parameters) many times. There is a general transformation from recursive algorithms to dynamic programming known as memoization, in which there is a table storing all results ever calculated by your recursive procedure. When the recursive procedure is called on a set of inputs which were already used, the results are just fetched from the table. This reduces recursive Fibonacci to iterative Fibonacci.

Dynamic programming can be even smarter, applying more specific optimizations. For example, sometimes there is no need to store the entire table in memory at any given time.

The dynamic programming method breaks this decision problem into smaller subproblems. Richard Bellman's principle of optimality describes how to do this:

> *Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*
>
> ***(See Bellman, 1957, Chap. III.3.)***

Solution to Question No. 2 Part B:

B 2.1 A naïve algorithm for the solution:

A naïve approach to this would be to multiply the two bit strings, bit-wise and then all of them together to get the product, this is a $O(n^2)$ time complexity since we have to iterate through every bit of the first bit string the second bit string number of times, if the length of the bit strings are n, n then that's $n \times n$ or n^2 times.

```
BIT-MULTIPLY(A, B, p, q)
   1. product = 0
   2. for i = 0 to q
   3.     partial_product = 0
   4.     for j = 0 to p
   5.         partial_product = B[i] * A[j]
   6.     product = product + (partial_product + i padded 0's)
   7. return product
```

Here we are assuming that "*" denotes bit multiplication and "+" denotes bit addition. These aren't that signification when considering the problem as a whole and hence has been omitted out. The time complexity is mostly dependent upon the length of the two strings.

B 2.2 A divide and conquer based algorithm:

In a divide and conquer based algorithm based solution for the given problem we split the bit string into two halves, the leftmost and the right most half. For the sake of simplicity we are assuming that the two bit strings have even length for now.

$$X = X_l \times 2^{\frac{n}{2}} + X_r$$

$$Y = Y_l \times 2^{\frac{n}{2}} + Y_r$$

where

X_l and X_r denote leftmost and rightmost $\frac{n}{2}$ bits of X

Y_l and Y_r denote leftmost and rightmost $\frac{n}{2}$ bits of Y

The product of XY can be written as the following

$$XY = \left(X_l \times 2^{\frac{n}{2}} + X_r\right)\left(Y_l \times 2^{\frac{n}{2}} + Y_r\right)$$

$$XY = 2^n X_l Y_l + 2^{\frac{n}{2}}(X_l Y_r + X_r Y_l) + X_r Y_r$$

The middle two terms can be further simplified to

$$X_l Y_r + X_r Y_l = (X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r$$

Hence the product now becomes

$$XY = 2^n X_l Y_l + 2^{\frac{n}{2}} \times [(X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r]$$

Now we take the case when the length of the strings isn't equal or aren't even. In these cases we put $\left\lfloor \frac{n}{2} \right\rfloor$ bits in the left half and $\left\lceil \frac{n}{2} \right\rceil$ bits in the right half. So the expression becomes.

$$XY = 2^{2\left\lceil \frac{n}{2} \right\rceil} X_l Y_l + 2^{\left\lceil \frac{n}{2} \right\rceil} \times [(X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r]$$

This algorithm is a standard algorithm by Karatsuba-Ofman, the algorithm is:

```
karatsuba(X, Y)
  1. if n== 1 then use multiplication table to find T = X*Y
  2. else split X, Y in half:
  3.     X = 2^(n/2)X1+X2
  4.     Y = 2^(n/2)Y1+Y2
  5.     Comment: X1, X2, Y1, Y2 each have n/2 bits.
  6.     U = karatsuba(X1, Y1)
  7.     V = karatsuba(X2, Y2)
  8.     W = Karatsuba(X1-X2, Y1-Y2)
  9.     Z = U + V - W
 10.     T = 2^nU+2^(n/2)Z+V
 11.     Comment: So U = X1*Y1, V = X2*Y2, W = (X1-X2)*(Y1-Y2), and
    therefore Z = X1 * Y2 + X2 * Y1. Finally we conclude that T =
    2^nX1 * Y1 + 2^(n/2)(X1 * Y2 + X2 * Y1) + X2 * Y2 = X * Y
 12.     return T
```

(Laszlo Babai, Algorithms CMSC-37000)

B 2.3 Analysis of time and space complexity of each one of the two algorithms:

Time complexity for Naive Algorithm:

The naïve solution has two nested loops, and is iterative in nature, the outer loop goes through each of the bit in the first number and the inner loop goes through all the bits in the second number, the partial product is obtained at every iteration and later is added to the final result.

Since we are going to each and every bit of the two numbers, the time complexity is $O(mn)$, where m is the length of the first number and n is the length of the second number in binary representation.

Space complexity for Naive Algorithm:

At every iteration we obtain the partial product which is saved in $2n$ space, and the overall product is saved in a cn space, hence the time complexity of the algorithm is $O(n)$.

Time complexity for Karatsuba Algorithm:

Since the algorithm multiplies two n-bit numbers, if $n = 2^k$ for some k, then the algorithm recurses three times on $\frac{n}{2} - bit$ number. Thus, the recurrence relation becomes

$$M(n) = 3 \times M\left(\frac{n}{2}\right)$$

And then considering the additions and subtractions, there are $O(n)$ additions and subtractions required for the algorithm. Therefore, the overall recurrence relation becomes:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

Using master theorem for solving the above recurrence relations, the time complexity turns out to be

$$O\left(n^{\log_2 3}\right) = O(n^{1.585})$$

Space complexity for Karatsuba Algorithm:

We can inductively say that the multiplying two n-length bit strings using Karatsuba Algorithm to be $O(cn) = O(n)$ space, here's an explanation of how,

$X1 * Y1$ can be computed in cn space and the result is saved in $2n$ space, then $X2 * Y2$ in cn space, the result of which is saved in $2n$ space, then $(X1 * Y1)(X2 * Y2)$ is computed in $(c + 4)n$ space and saved in $4n$ space. Hence it takes only $O(n)$ space to compute the product of two n length bit string numbers.

B 2.4 Conclusion:

This paradigm, divide-and-conquer, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. Divide the problem into a number of subproblems that are smaller instances of the same problem.
2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. Combine the solutions to the subproblems into the solution for the original problem.

Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.

In this question we can clearly see that divide and conquer is more efficient way of solving the same problem, the only condition for divide and conquer to work in our case is that we were able to divide the multiplication of the two bit strings into multiplication of two halves of the strings itself, which is further divided until we hit the base case.

1. Laszlo Babai, Divide and Conquer: The Karatsuba-Ofman algorithm, Algorithms – CMSC-37000
2. ICS 161, Lecture Notes, https://www.ics.uci.edu/~eppstein/161/960229.html
3. https://www.educative.io/collection/page/10370001/760001/1380003
4. Prof Jozef Gruska, DrSc, FUTURE in INFORMATICS ERA 2015
5. https://towardsdatascience.com/an-overview-of-monte-carlo-methods-675384eb1694