CS 590 - Assignment 4

Submitted By: Shweta Madhale

CWID: 20015921

Abstract:

Dynamic programming is an approach wherein the problem is solved by combining the solutions for subproblems. This approach is useful when the subproblems are overlapping. It solves each subproblem and stores it in tabular format. So, it does not recompute the same problem repeatedly but just revisits the value in the table. Dynamic programming can be used for various applications like sequence alignment, longest common subsequence, etc. Implementation of the Smith-Waterman Algorithm for sequence alignment was done using C++. This implementation was done in a bottom-up and top-down(memoization) manner. Also, the maximum alignment was obtained with each new sequence. Understanding the greedy approach and dynamic approach was done with some problems.

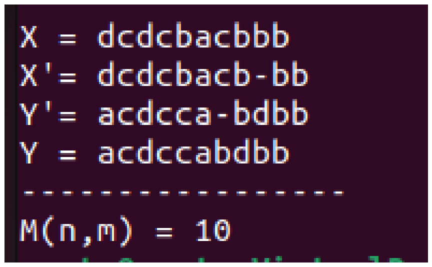
Observations:

Q.1-3

Analysis and Conclusion:

i. The Smith-Waterman Algorithm can be implemented using two approaches viz. bottom-up and top-down. The bottom-up approach is an iterative problem while the top-down use memoization recursively.
ii. The bottom-up is the optimal approach out of the two. It follows the basic concept of dynamic programming and continues by finding optimal solutions for the subproblems.
iii. The top-down approach is not as efficient as it has an overhead for recursion.
iv. The top-down approach can be useful for cases with memory or time constraints, as it only computes the required subproblems.
v. For this problem, of sequence alignment using Smith-Waterman Algorithm, the bottom-up approach has an advantage over the top-down.

Q.4



```
X = dcdcbacbbb
X'= dcdcbacb-bb
Y'= acdcca-bdbb
Y = acdccabdbb
-----------------
M(n,m) = 10
```

H Table

	-	d	c	d	c	b	a	c	b	b	b
-	0	0	0	0	0	0	0	0	0	0	0
a	0	**-1**	**-1**	**-1**	**-1**	**-1**	**2**	**1**	**0**	**-1**	**-1**
c	0	**-1**	**1**	**0**	**1**	**0**	**1**	**4**	**3**	**2**	**1**
d	0	**2**	**1**	**3**	**2**	**1**	**0**	**3**	**3**	**2**	**1**
c	0	**1**	**4**	**3**	**5**	**4**	**3**	**2**	**2**	**2**	**1**
c	0	**0**	**3**	**3**	**5**	**4**	**3**	**5**	**4**	**3**	**2**
a	0	**-1**	**2**	**2**	**4**	**4**	**6**	**5**	**4**	**3**	**2**
b	0	**-1**	**1**	**1**	**3**	**6**	**5**	**5**	**7**	**6**	**5**
d	0	**2**	**1**	**3**	**2**	**5**	**5**	**4**	**6**	**6**	**5**
b	0	**1**	**1**	**2**	**2**	**4**	**4**	**4**	**6**	**8**	**8**
b	0	**0**	**0**	**1**	**1**	**4**	**3**	**3**	**6**	**8**	**10**

P Table

	-	d	c	d	c	b	a	c	b	b	b
-	0	0	0	0	0	0	0	0	0	0	0
a	0	**d**	**d**	**d**	**d**	**d**	**d**	**a**	**a**	**d**	**d**
c	0	**d**	**d**	**a**	**d**	**a**	**l**	**d**	**a**	**a**	**a**
d	0	**d**	**a**	**d**	**a**	**a**	**a**	**l**	**d**	**d**	**d**
c	0	**l**	**d**	**a**	**d**	**a**	**a**	**d**	**d**	**d**	**d**
c	0	**l**	**d**	**d**	**d**	**d**	**d**	**d**	**a**	**a**	**a**
a	0	**d**	**l**	**d**	**l**	**d**	**d**	**a**	**d**	**d**	**d**
b	0	**d**	**l**	**d**	**l**	**d**	**a**	**d**	**d**	**d**	**d**
d	0	**d**	**a**	**d**	**a**	**l**	**d**	**d**	**l**	**d**	**d**
b	0	**l**	**d**	**l**	**d**	**d**	**d**	**d**	**d**	**d**	**d**
b	0	**l**	**d**	**l**	**d**	**d**	**d**	**d**	**d**	**d**	**d**

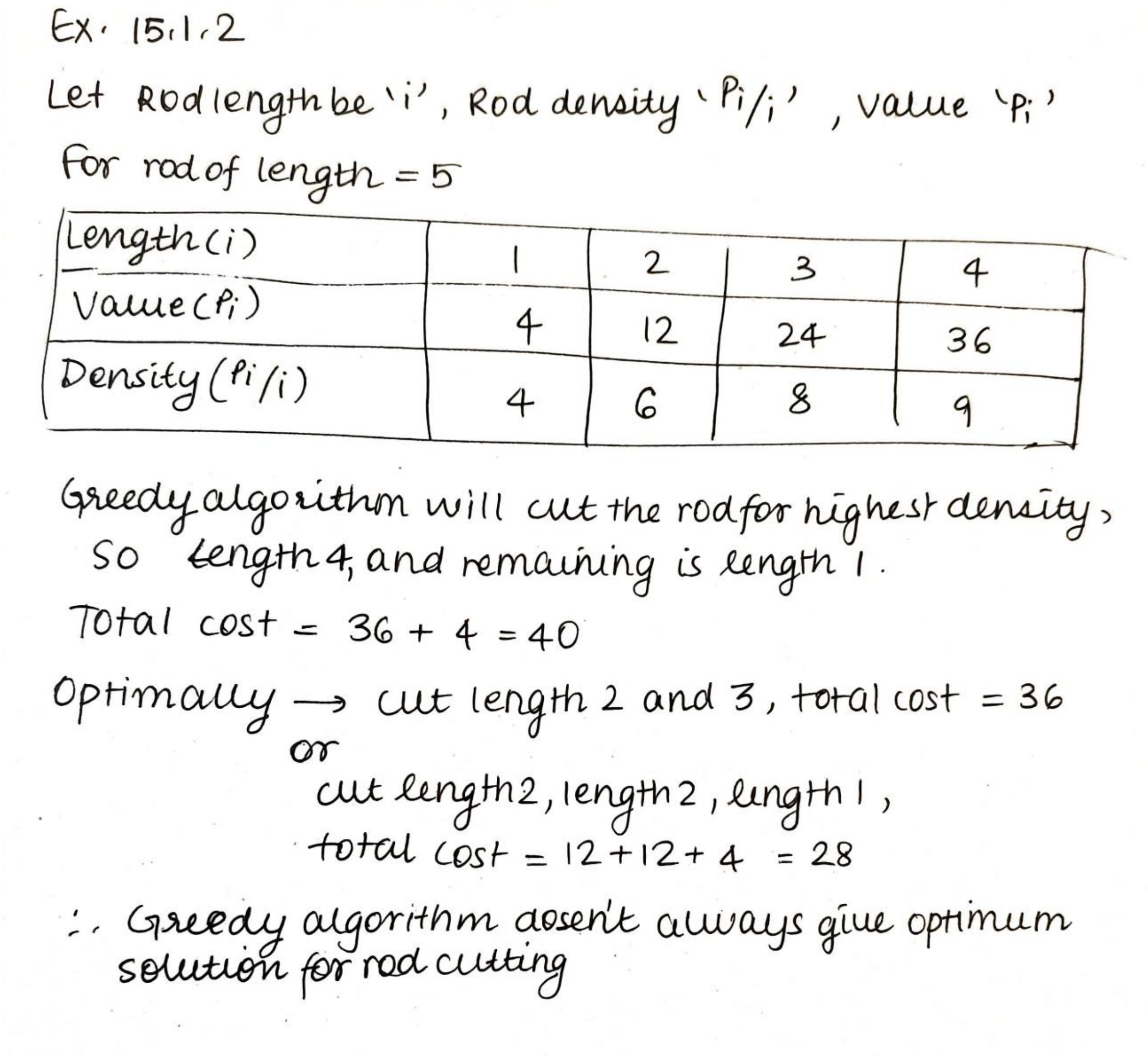
So, the result is.

X'= ‘dcdcbacb-bb’

Y'= ‘acdcca-bdbb’

Maximum Alignment = 10

Q.15.1-2



Ex. 15.1.2

Let Rod length be ‘i’, Rod density ‘p_i/i’, value ‘p_i’

For rod of length = 5

Length(i)	1	2	3	4
Value(p_i)	4	12	24	36
Density(p_i/i)	4	6	8	9

Greedy algorithm will cut the rod for highest density,
So length 4, and remaining is length 1.

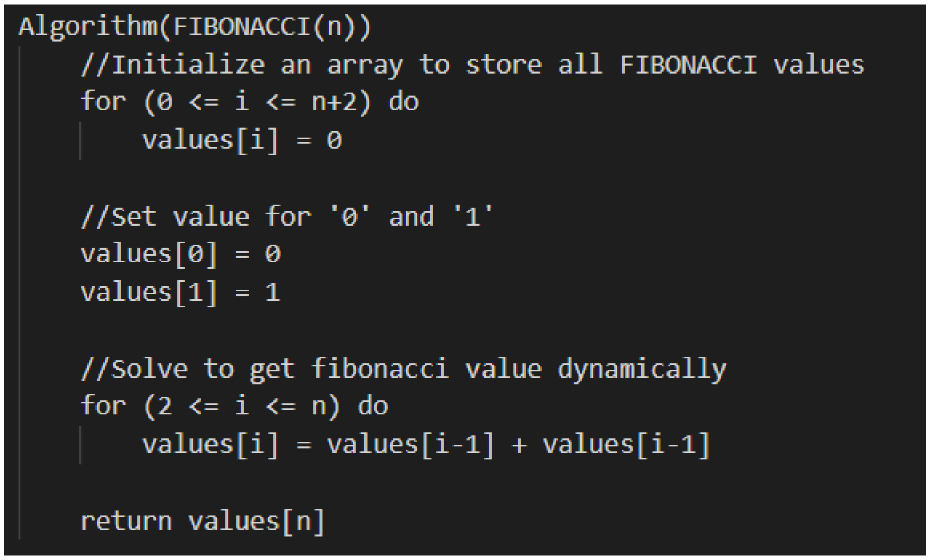
Total cost = 36 + 4 = 40

Optimally $\longrightarrow$ cut length 2 and 3, total cost = 36
or
cut length 2, length 2, length 1,
total cost = 12 + 12 + 4 = 28

$\therefore$ Greedy algorithm doesn't always give optimum solution for rod cutting

Q.15.1-5

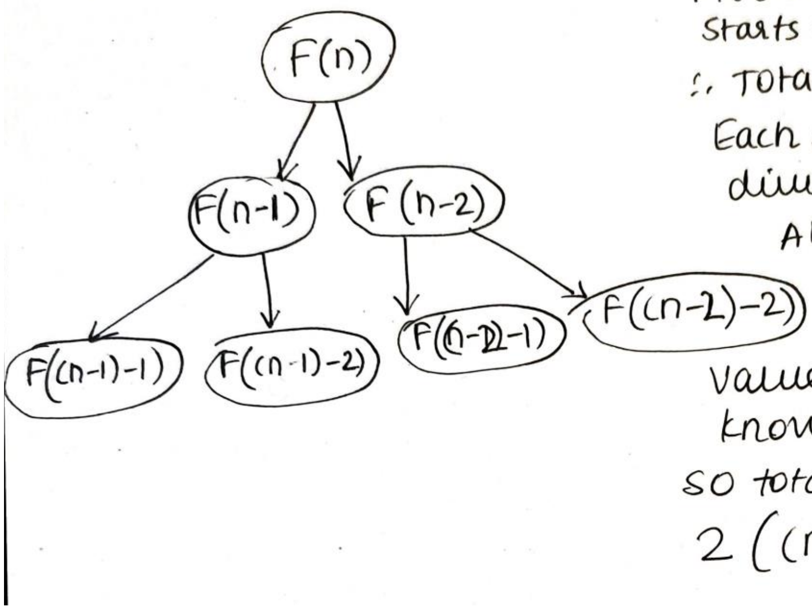
a. Pseudocode for the Fibonacci problem



```
Algorithm(FIBONACCI(n))
    //Initialize an array to store all FIBONACCI values
    for (0 <= i <= n+2) do
        values[i] = 0

    //Set value for '0' and '1'
    values[0] = 0
    values[1] = 1

    //Solve to get fibonacci value dynamically
    for (2 <= i <= n) do
        values[i] = values[i-1] + values[i-1]

    return values[n]
```

b. Graph of subproblems, count of vertices and edges

Ex-15.1-5



Fibonacci sequence starts with 0 - - - - n

∴ Total vertices = n+1

Each subproblem is divided in 2 subproblems,

Also no. of subproblems is n+1,

Value of F(0) and F(1) is known,

so total edges =

$2((n+1)-2) = 2(n-1)$

Q.15.4-1

	-	1	0	0	1	0	1	0	1
-	0	0	0	0	0	0	0	0	0
0	0	**0↑**	**1↖**	**1↖**	**1←**	**1↖**	**1←**	**1↖**	**1←**
1	0	**1↖**	**1↑**	**1↑**	**2↖**	**2↑**	**2↖**	**2↑**	**2↖**
0	0	**1←**	**2↖**	**2↖**	**2↑**	**3↖**	**3↑**	**3↖**	**3↑**
1	0	**1↖**	**2←**	**2↑**	**3↖**	**3↑**	**4↖**	**4↑**	**4↖**
1	0	**1↖**	**2←**	**2↑**	**3↖**	**3↑**	**4↖**	**4↑**	**5↖**
0	0	**1←**	**2↖**	**3↖**	**3↑**	**4↖**	**4↑**	**5↖**	**5↑**
1	0	**1↖**	**2←**	**3←**	**4↖**	**4↑**	**5↖**	**5↑**	**6↖**
1	0	**1↖**	**2←**	**3←**	**4↖**	**4↑**	**5↖**	**5↑**	**6↖**
0	0	**1←**	**2↖**	**3↖**	**4←**	**5↖**	**5↑**	**6↖**	**6↑**

Therefore, the LCS is <1, 0, 1, 0, 1, 0>