**Part 1:**

1. **Bottom up version:**

int\*\* bottomupSW(char\* X, char\* Y, int n, int m, int\*\* H, int\*\* P)

{

int i, j;

int p1, p2, p3;

for(i=0; i<n; i++)

{

H[i][0] = P[i][0] = 0;

}

for(j=0; j<m; j++)

{

H[0][j] = P[0][j] = 0;

}

for(i=0; i<n; i++)

{

for(j=0; j<m; j++)

{

if(X[i] == Y[j])

p1 = H[i][j] + 2;

else

p1 = H[i][j] -1;

p2 = H[i][j+1] -1;

p3 = H[i+1][j] -1;

//H[i+1][j+1] = max(p1,p2,p3);

if(p1>=p2 && p1>=p3)

H[i][j]=p1;

else if(p2>=p1 && p2>=p3)

H[i][j]=p2;

else if(p3>=p1 && p3>=p2)

H[i][j]=p3;

if(H[i+1][j+1] == p1)

P[i+1][j+1] = '@';

else

{

if(H[i+1][j+1] == p2)

P[i+1][j+1] = '|';

else

{

P[i+1][j+1] = '#';

}

}

}

}

return H;

}

1. **Top down with memorization:**

int topdownSW(char\* X, char\* Y, int m, int n, int\*\* ops)

{

if (m == 0 || n == 0)

return 0;

if (ops[m-1][n-1] != INT\_MIN)

return ops[m-1][n-1];

if (X[m-1] == Y[n-1]) {

ops[m-1][n-1] = 2 + topdownSW(X, Y, m - 1, n - 1,ops);

return ops[m-1][n-1];

}

else {

int t = max((topdownSW(X, Y, m, n - 1,ops)-1),

(topdownSW(X, Y, m - 1, n,ops)-1));

ops[m-1][n-1]=max(t,topdownSW(X, Y, m - 1, n - 1,ops)-1);

return ops[m-1][n-1];

}

}

1. **Print-Seq-Align-X and Print-Seq-Align-Y:**

void printSeqAlignX(char\* X,int\*\* P,int n,int m)

{

if(P[n][m] == '@')

{

printSeqAlignX(X,P,n-1,m-1);

cout << X[n-1];

}

else

{

if (P[n][m] == '#')

{

printSeqAlignX(X,P,n,m-1);

cout << "-";

}

else

{

printSeqAlignX(X,P,n-1,m);

cout << X[n-1];

}

}

}

void printSeqAlignY(char\* Y,int\*\* P,int n,int m)

{

if(P[n][m] == '@')

{

printSeqAlignY(Y,P,n-1,m-1);

cout << Y[n];

}

else

{

if (P[n][m] == '#')

{

printSeqAlignY(Y,P,n,m-1);

cout << "-";

}

else

{

printSeqAlignY(Y,P,n-1,m);

cout << Y[n];

}

}

}

1. Find the maximum alignment for  **X=dcdcbacbbb and Y=acdccabdbb** by using Smith-Waterman algorithm. Execute the pseudocode algorithm and fill the necessary tables H and P in a bottom-up fashion. Reconstruct the strings X’ and Y’ using the tables H and P.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X/Y |  | A | C | D | C | C | A | B | D | B | B |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 0 | -1 | -1 | 2 | 1 | 0 | -1 | -1 | 2 | 1 | 0 |
| C | 0 | -1 | 1 | 1 | 4 | 3 | 2 | 1 | 1 | 1 | 0 |
| D | 0 | -1 | 0 | 3 | 3 | 3 | 2 | 1 | 3 | 2 | 1 |
| C | 0 | -1 | 1 | 2 | 5 | 5 | 4 | 3 | 2 | 2 | 1 |
| B | 0 | -1 | 0 | 1 | 4 | 4 | 4 | 6 | 5 | 4 | 4 |
| A | 0 | 2 | 1 | 0 | 3 | 3 | 6 | 5 | 5 | 4 | 3 |
| C | 0 | 1 | 4 | 3 | 2 | 5 | 5 | 5 | 4 | 4 | 3 |
| B | 0 | 0 | 3 | 3 | 2 | 4 | 4 | 7 | 6 | 6 | 6 |
| B | 0 | -1 | 2 | 2 | 2 | 3 | 3 | 6 | 6 | 8 | 8 |
| B | 0 | -1 | 1 | 1 | 1 | 2 | 2 | 5 | 5 | 8 | 10 |
|  |  |  | **C** | **D** | **C** | **C** |  | **B** |  | **B** | **B** |

**Solution**: <CDCCBBB>.

**Part 2:**

1. Show, by means of a counter example, that the following “greedy” strategy does not always determine an optimal way to cut rods. Define the **density** of a rod of length*i* to be *pi/i,* that is, its value per inch. The greedy strategy for a rod of length *n* cuts off a first piece of length *i,* where 1 <= I <= n, having maximum density. It then continues by applying the greedy strategy to be remaining piece of length *n-i.*

Counter example for greedy strategy; Let n be 5 which will be the length of the rod.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **i 🡪 number of cuts** | **1** | **2** | **3** | **4** | **5** |
| **pi 🡪 price per inch** | 2 | 20 | 33 | 36 | 40 |
| **pi/i 🡪 density** | 2 | 10 | 11 | 9 | 8 |
| **Solution** | 5(1) \* 2 – 10 | 2(2) +1 – 42 | 3(1) + 1+1 -37  3(1) + 2(1) - 53 | 4(1)+1-38 | 5 – 40  4+1 – 38  3+2 – 53  2+2+1 – 42 |
| **Optimal Solution** |  |  |  |  | **3+2=53** |

According to greedy strategy, without cutting the rod, the total value would be 40. If we cut the rod into two, 4 inch and 1 inch, the value would be 38. If we cut the rod, 3 inch and 2 inches, the value would be 53. If it is cut into 3, 2-inch, 2 inch and 1 inch, value would be 42. **Hence the optimal solution is 53.**

1. The Fibonacci numbers are defined by recurrence(3.22). Give an *O(n)* time dynamic-programming algorithm to compute the n-th Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

Fibonacci(n)

Let fib(0….n) be a new array

fib(0) = fib(1) =1

For i = 2 to n

fib(i) = fib(i-1) + fib(i-2)

Return fib(n)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 4 |  |  |  |
|  |  | 3 |  |  | 2 |  |
|  | 2 |  | 1 | 0 |  | 1 |
| 0 |  | 1 |  |  |  |  |

The number of vertices in the tree will follow the recurrences. Each number in the sequence is sum of two previous numbers in the sequence.

V(n) = 1+ v(n-2) +v(n-1).

The initial conditions are v(0) = v(1) =1.

The above graph shows that

V(n) = 1+ (2 \* fib(n-2) – 1) + (2\* fib(n-1)-1) = 2 \* fib(n) -1

Thus, subproblem graph consists of n+1 vertex.

The number of edges will satisfy the recurrence;

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

And the base cases are E(0) = E(1) =0.

By induction,

E(n) = 2\*fib(n) - 2

Thus, the subproblem graph has 2n – 2 edges.

1. Determine an LCS of (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0).

The selected cell is shaded with grey colour. The LCS is <010101>.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S** |  | **0** | **1** | **0** | **1** | **1** | **0** | **1** | **1** | **0** |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **0** | **0** | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| **0** | 0 | **1** | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| **1** | 0 | 1 | **2** | 2 | 3 | 3 | 3 | 4 | 4 | 4 |
| **0** | 0 | 1 | 2 | **3** | **3** | 3 | 4 | 4 | 4 | 5 |
| **1** | 0 | 1 | 2 | 3 | 4 | **4** | 4 | 5 | 5 | 5 |
| **0** | 0 | 1 | 2 | 3 | 4 | 4 | **5** | **5** | 5 | 6 |
| **1** | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 6 | **6** | **6** |
| **LCS** |  | **0** | **1** | **0** |  | **1** | **0** |  | **1** |  |