SC2001/CX2101
Algorithm Design and Analysis

Tutorial 4
Dynamic Programming

(Weeks 10-11)

This tutorial helps you develop skills in the learning outcome of the course: "Able to design algorithms using suitable strategies (dynamic programming, etc) to solve a problem, able to analyse the efficiencies of different algorithms for problems like optimal sequencing for matrix multiplication, the longest common subsequence, etc".

Question 1

Find the length of the longest common subsequence and a longest common subsequence of CAGAG and ACTGG by the dynamic programming algorithm in the lecture notes.

	1	2	3	4	5
x	C	A	G	A	G
y	A	C	T	G	G

c		A	C	T	G	G
	0	0	0	0	0	0
C	0	0	1	1	1	1
A	0	1	1	1	1	1
G	0	1	1	1	2	2
A	0	1	1	1	2	2
G	0	1	1	1	2	3

h		A	C	T	G	G
	—	—	—	—	—	—
C	|	|	\	—	—	—
A	|	\	|	|	|	|
G	|	|	|	|	\	\
A	|	\	|	|	|	|
G	|	|	|	|	\	\

```
for i = 1 to n
   for j = 1 to m
      if x[i] == y[j]  {
         c[i][j] = c[i-1][j-1] + 1;
         h[i][j] = '\';   }
      else if c[i-1][j] >= c[i][j-1] {
         c[i][j] = c[i-1][j];
         h[i][j] = '|';   }
      else {
         c[i][j] = c[i][j-1];
         h[i][j] = '—';   }
```

LCS(5,5) = 3

<table>
<tr><th>h</th><th></th><th>A</th><th>C</th><th>T</th><th>G</th><th>G</th></tr>
<tr><td></td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td><td>—</td></tr>
<tr><td>C</td><td>|</td><td>|</td><td>\</td><td>—</td><td>—</td><td>—</td></tr>
<tr><td>A</td><td>|</td><td>\</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>|</td><td>|</td><td>\</td><td>\</td></tr>
<tr><td>A</td><td>|</td><td>\</td><td>|</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>G</td><td>|</td><td>|</td><td>|</td><td>|</td><td>\</td><td>\</td></tr>
</table>

The subsequence: **C** **G** **G**

Question 2

The H-number H(n) is defined as follows:

H(0) =1, and for n>0:

$H(n) = H(n-1) + H(n-3) + H(n-5)+ \ldots.+H(0)$ when n is odd

$H(n) = H(n-2) + H(n-4) + H(n-6)+ \ldots.+H(0)$ when n is even.

a) Give a recursive algorithm to compute H(n) for an arbitrary n as suggested by the recurrence equation given for H(n). Draw the tree that represents the recursive calls made when H(8) is computed.

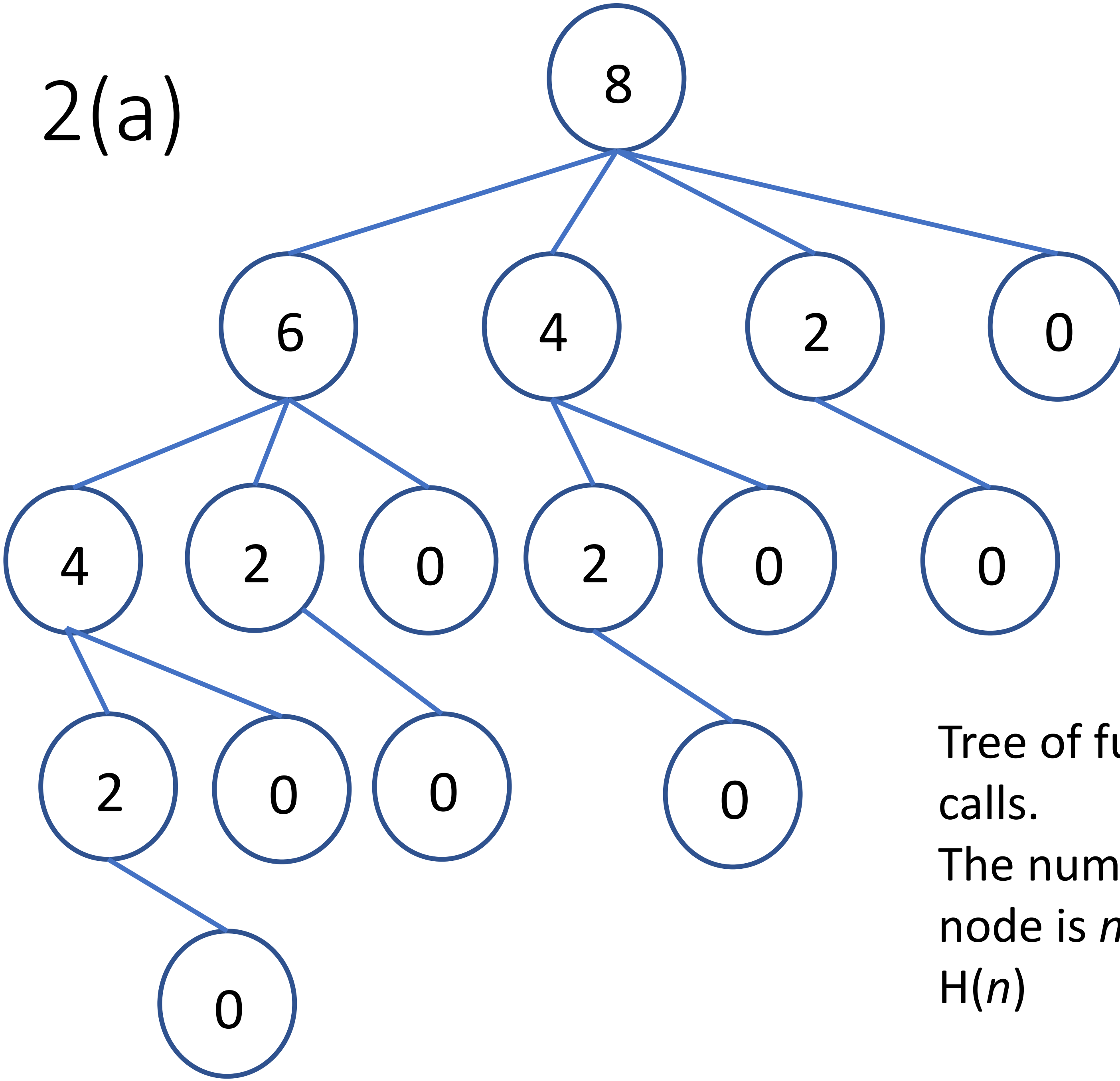
b) Draw the subproblem graph for H(8) and H(9).

c) Write an iterative algorithm using the dynamic programming approach (bottom-up). What are the time and space required?

Question 2(a)

```
int  hn( int n) {
{    if (n == 0)    return 1;
     else {
          S = 0;
          if (n mod 2)     j=n-1;      else  j=n-2;
          for (k = 0; k <= j; k = k+2)
               S += hn(k);
      }
      return S;
}
```

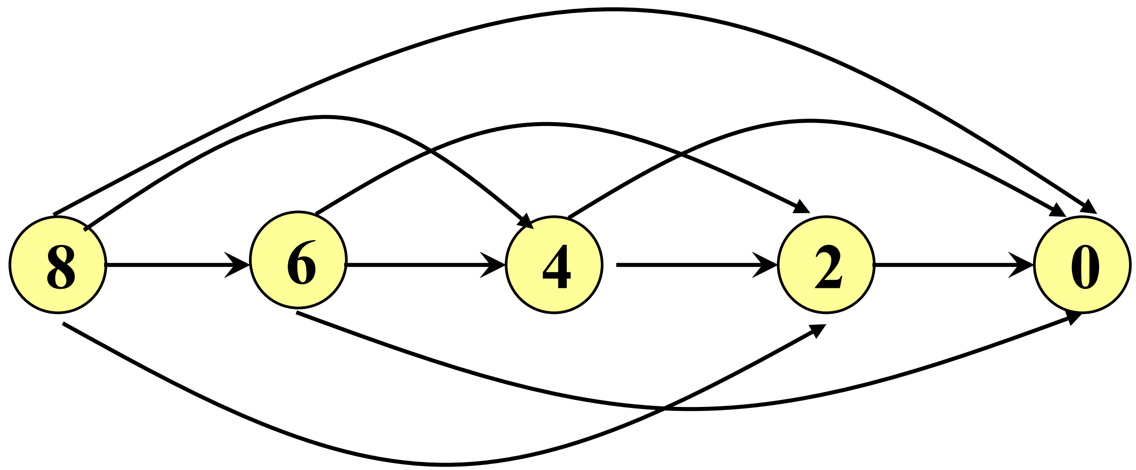
2(a)



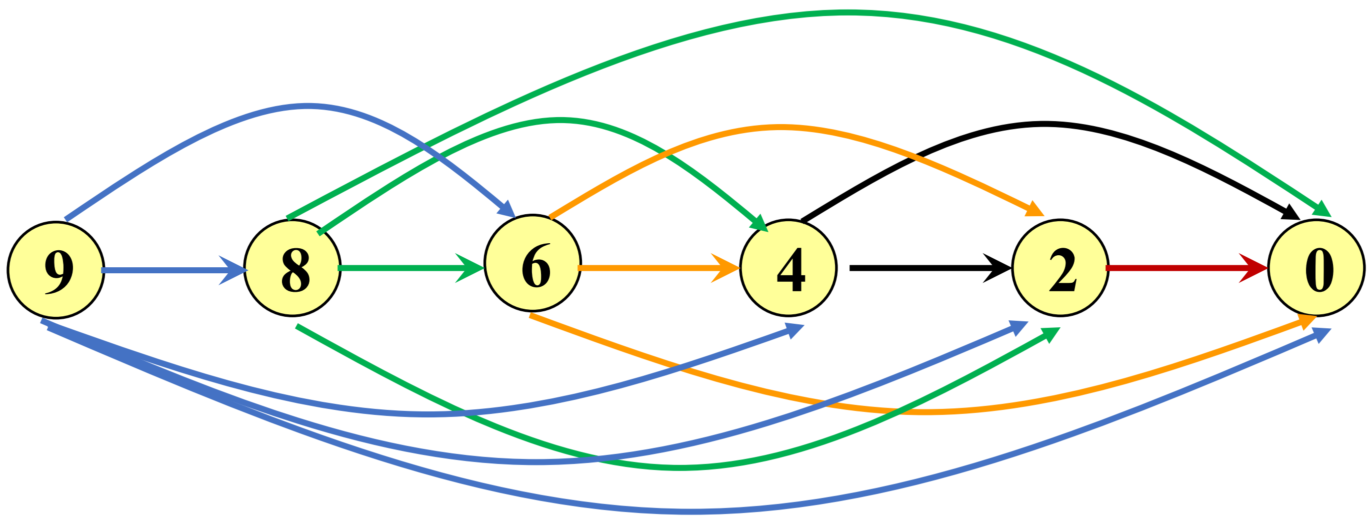
Tree of function calls.
The number in a node is *n* for a call H(*n*)

2(b)

The subproblem graph for H(8)



The subproblem graph for H(9)



2(c)

```
int  hn_DP(int n)
{     // Make use of an array S[0..n]
      S[0]=1;
      for (i = 1; i<=n; i++) {
            S[i] = 0;
            if (i mod 2)    j = i-1;    else  j=i-2;
            while (j>=0) { S[i] += S[j]; j-=2;};
       }
       return S[n];
}
```

Space Complexity: $O(n)$.   Time complexity: $O(n^2)$

Question 3

The binomial coefficients can be defined by the recurrence equation:

$C(n, k) = C(n-1, k-1) + C(n-1, k)$	for $n > 0$ and $k > 0$
$C(n, 0) = 1$	for $n >= 0$
$C(0, k) = 0$	for $k > 0$

$C(n, k)$ is also called "*n* choose *k*". This is the number of ways to choose *k* distinct objects from a set of *n* objects.

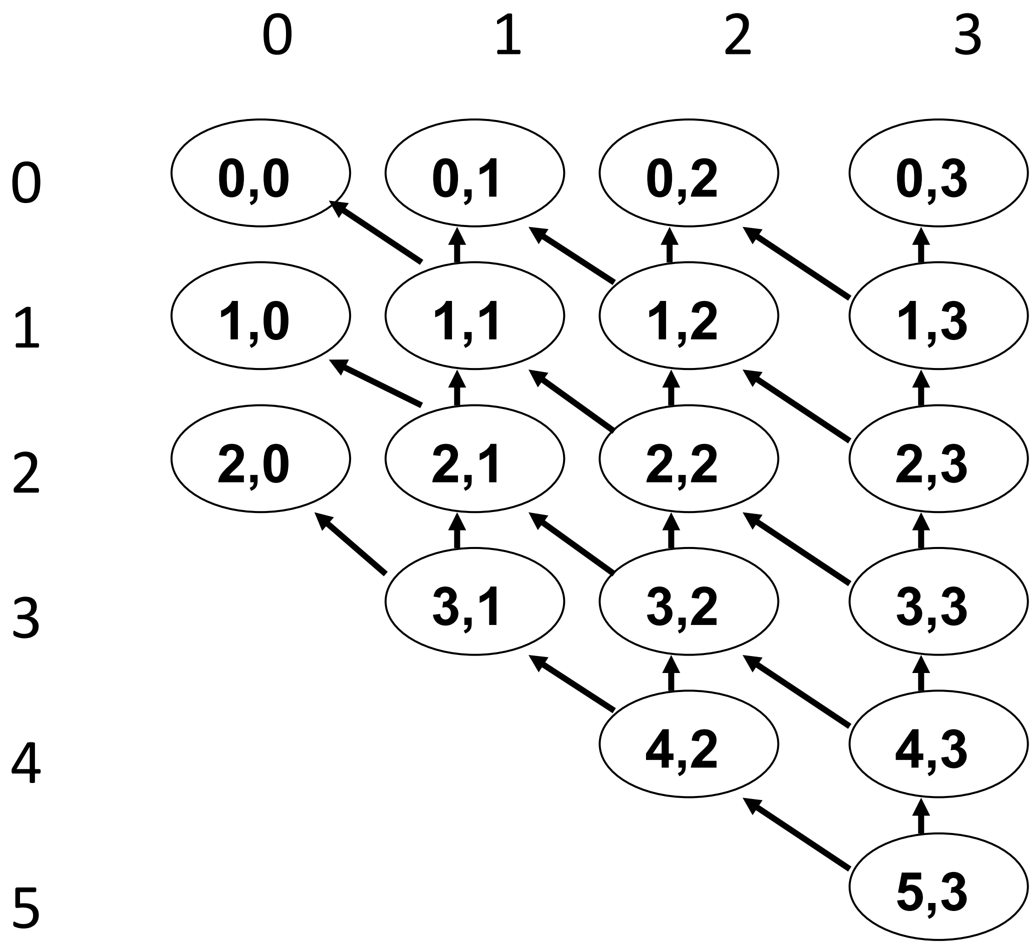
3(a)

Give a recursive algorithm as suggested by the recurrence equation given for C(*n*, *k*).

```
int C(int n, int k)
{
        if (k == 0)   return 1;
        if (n == 0) return 0;

        return C(n – 1, k – 1) + C(n – 1, k);
}
```

3(b) Draw the subproblem graph for C(5, 3).



3(c)

Write a recursive algorithm using the dynamic programming approach (top-down) stating the data structure used for the dictionary.

Use dictionary: int dic[n+1][k+1];
// initialised to –1 in all entries

```
int C(int n, int k, int [] [] dic)
{     int c1, c2;

      if (k == 0)   {
            dic[n][0] = 1;
            return 1;  }
      if (n == 0) {
            dic[0][k] = 0;
            return 0;   }

      if (dic[n – 1][k – 1] == -1)
            c1 = C(n – 1, k – 1);
      else c1 = dic[n – 1][k – 1] ;
      if (dic[n – 1][k] == -1)
            c2 = C(n – 1, k);
      else c2 = dic[n – 1][k] ;

      dic[n][k] = c1 + c2;
      return dic[n][k];
}
```

Time complexity: O(nk)
Space complexity: O(nk)

3(d)

Write an iterative algorithm using the dynamic programming approach (bottom-up).

```
int C(int n, int k, int [] [] dic)
{    int dic[n+1][k+1];

     For (i = 1;  i<= k;  i++)  dic[0][i] = 0;
     For (i = 0;  i<= n;  i++)  dic[i][0] = 1;
     For (i = 1;  i<= n;  i++)
          For (j = 1;  j<= k;  j++)
               dic[i][j] = dic[i-1][j-1] + dic[i-1][j];

    Return dic[n][k];
}
```

Time complexity: O(nk)
Space complexity: O(nk)

```
int C(int n, int k, int [] [] dic)  // more optimized
{     int dic[n+1][k+1];

      For (i = 1;  i<= k;  i++)  dic[0][i] = 0;
      For (i = 0;  i<= n-k;  i++)  dic[i][0] = 1;
      For (i = 1;  i<= n;  i++)
            For (j = max(i-(n-k), 1);  j<= k;  j++)
                  dic[i][j] = dic[i-1][j-1] + dic[i-1][j];

      Return dic[n][k];
}
```

Question 4

Construct an example with only three or four matrices where the worst multiplication order does at least 100 times as many element-wise multiplications as the best order.

Let the dimensions of A, B and C be 100x1, 1x100, 100x1 respectively.
Best order: A(BC) – the no. of multiplications is 200
Worst order: (AB)C – the no. of multiplications is 20000

Question 5

Suppose the dimensions of the matrices *A*, *B*, *C*, and *D* are 20x2, 2x15, 15x40, and 40x4, respectively, and we want to know how best to compute *A*x*B*x*C*x*D*. Show the arrays **cost** and **last** computed by Algorithms matrixOrder() in the lecture notes.

Array **d**

20	**2**	**15**	**40**	**4**
0	**1**	**2**	**3**	**4**

Cost

	0	1	2	3	4
0		0	600		
1			0	1200	
2				0	2400
3					0
4					

Last

	0	1	2	3	4
0			1		
1				2	
2					3
3					
4					

Array **d**

20	2	15	40	4

Cost[0][2] = Cost[0][1] +Cost[1][2] + d[0]*d[1]*d[2]

Last[0][2] =1

Cost[1][3] = Cost[1][2] +Cost[2][3] + d[1]*d[2]*d[3]

Last[1][3] =2

Cost[2][4] = Cost[2][3] +Cost[3][4] + d[2]*d[3]*d[4]

Last[2][4] =3

Cost

	0	1	2	3	4
0		0	600	2800	
1			0	1200	1520
2				0	2400
3					0
4					

Last

	0	1	2	3	4
0			1	1	
1				2	3
2					3
3					
4					

Array **d**

20	2	15	40	4

Cost[0][3] = min(Cost[0][1]
+Cost[1][3] + d[0]*d[1]*d[3],
Cost[0][2] +Cost[2][3] +
d[0]*d[2]*d[3])
=min(1200+1600, 600+12000)
=2800

Last[0][3] =1

Cost[1][4] = min(Cost[1][2]
+Cost[2][4] + d[1]*d[2]*d[4],
Cost[1][3] +Cost[3][4] +
d[1]*d[3]*d[4])
=min(2400+120, 1200+320)
=1520

Last[1][4] =3

Cost

	0	1	2	3	4
0		0	600	2800	1680
1			0	1200	1520
2				0	2400
3					0
4					

Last

	0	1	2	3	4
0			1	1	1
1				2	3
2					3
3					
4					

Array **d**

20	2	15	40	4

Cost[0][4] = min (Cost[0][1] +Cost[1][4] + d[0]*d[1]*d[4], Cost[0][2] +Cost[2][4] + d[0]*d[2]*d[4], Cost[0][3] +Cost[3][4] + d[0]*d[3]*d[4])
=min(1520+160, 600+2400+1200, 2800+3200)
=1680

Last[0][4] =1

Question 6

We have a knapsack of size 10 and 4 objects. The sizes and the profits of the objects are given by the table below. Find a subset of the objects that fits in the knapsack that maximizes the total profit by the dynamic programming algorithm in the lecture notes.

p	10	40	30	50
s	5	4	6	3

C = 10

p	10	40	30	50
w	5	4	6	3

profit

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	50
4	0	0	40	40	50
5	0	10	40	40	50
6	0	10	40	40	50
7	0	10	40	40	90
8	0	10	40	40	90
9	0	10	50	50	90
10	0	10	50	70	90

```
for r = 1 to C
    for c = 1 to n
        profit[r][c] = profit[r][c-1];
        if (w[c] <= r)
            if (profit[r][c] <
        profit[r-w[c]][c-1] + p[c])
                profit[r][c] =
                    profit[r-w[c]][c-1]
                    + p[c]
```

Question 7

S1 is a sequence of *n1* characters and *S2* is a sequence of *n2* characters. All characters are from the set {'a', 'c', 'g', 't'}. An alignment is defined by inserting any number of character '_' (the underscore character) into *S1* and *S2* so that the resulting sequences *S1'* and *S2'* are of equal length. Each character in *S1'* has to be aligned with the same character or an underscore in the same position in *S2'* and vice versa. The cost of an alignment of *S1* and *S2* is defined as the number of underscore characters inserted in *S1* and *S2*. For example, *S1* = "ctatg" and *S2* = "ttaagc". One possible alignment is

S1' = "ct_at_g_" and

S2' = "_tta_agc"

Both *S1'* and *S2'* have length 8 and the cost is 5. We want to find the minimum cost of aligning two sequences, denoted as alignment(*n1*, *n2*).

Question 7

a) Give a recursive definition of alignment(*n1*, *n2*).

Analysis:

Two base cases: (i) *S2* is empty then *S2*’ has n1 ‘_’ characters; (ii) *S1* is empty then *S1*’ has n2 ‘_’ characters.

When both *S1* and *S2* are not empty, we have two possibilities:
(i) S1[n1] == S2[n2], no insertion, the last character of *S1*’ and *S2*’ is this character, preceded by the best alignment from Alignment(n1-1, n2-1)

(ii) S1[n1] ≠ S2[n2], we may align the last character of *S1* with ‘_’ and find Alignment(n1-1, n2), we may also align the last character of *S2* with ‘_’ and find Alignment(n1, n2-1). In both ways, we have one ‘_’ insertion. The minimum cost is the minimum between these two ways.

Question 7

a) Give a recursive definition of alignment(*n1*, *n2*).

Alignment(n1, 0) = n1

Alignment(0, n2) = n2

Alignment(n1, n2)

```
    = Alignment(n1-1, n2-1)      // if S1[n1] == S2[n2],
    = min(Alignment(n1-1, n2), Alignment(n1, n2-1)) + 1
        //  otherwise
```

Question 7

b) Draw the subproblem graph for alignment(3, 4).

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)

For each (x, y) where $x \neq 0$ and $y \neq 0$, it has an edge to (x-1, y-1), (x-1, y) and (x, y-1)

Question 7

c) Design a dynamic programming algorithm of alignment(*n1*, *n2*) using the bottom-up approach.

```
For (r = 0 to n1)  cost[r][0] = r;
For (c = 1 to n2)  cost[0][c] = c;
For (r = 1 to n1)
    For (c = 1 to n2)
        If  (S1[r] == S2[c])  cost[r][c] = cost[r-1][c-1];
        Else if (cost[r-1][c] < cost[r][c-1])
            cost[r][c] = cost[r-1][c] + 1;
        Else cost[r][c] = cost[r][c-1] + 1;
Return cost[n1][n2];
```