1 Fibonacci numbers

One of the most famous sequences of numbers is the Fibonacci sequence: $1, 1, 2, 3, 5, 8, 13, 21, 34, ...$ The sequence is defined recursively:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

The Fibonacci numbers have various interesting properties, including being related to the golden ratio and its conjugate:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$$

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.61803$$

You could prove by induction that $F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$.

Here is an obvious algorithm for computing $F(n)$, the n-th Fibonnaci number:

```
Fib(n)
if n = 0 or n = 1
  return 1
else
  return Fib(n-1) + Fib(n-2)
```

The running time $T(n)$ of this algorithm on input n is?

$$T(n) = T(n-1) + T(n-2) + \Theta(1).$$

Consider the tree of the calls made during the computation of Fib(n):

```
                        Fib n
                      /       \
                     /         \
                    /           \
             Fib n-1             Fib n-2
            /       \           /       \
           /         \         /         \
      Fib n-2     Fib n-3  Fib n-3    Fib n-4
     /       \   /      \  /      \  /       \
    /         \
Fib n-3    Fib n-4
/     \    /     \
```

Note that the shallowest node of this tree is the "rightmost" node and it has depth $\frac{n}{2}$. Therefore, the total number of recursive calls is in $\Omega(2^{\frac{n}{2}})$. In fact, the solution to the above recursion is:

$$T(n) = \Theta(\phi^n).$$

which is exponential. The reason our recursive algorithm is so slow is because of the same recursive calls are recomputed over and over (note that Fib(n-3), for example, is computed 3 times).

We would like a better strategy for computing the Fibonacci numbers, one that does not duplicate so much of the computation. We can achieve that if we take a bottom-up approach, rather than a top-down approach. For example, to compute $F(n)$ we should start by computing $F(2)$ from $F(0) = 1$ and $F(1) = 1$. Then we compute $F(3)$ from $F(2)$ and $F(1)$, and then $F(4)$ from $F(3)$ and $F(2)$, and so on. Here is a much faster algorithm:

```
FastFib(n)
F[0] = 1
F[1] = 1
for i = 2 to n do
  F[i] = F[i-1] + F[i-2]
return F[n]
```

The running time of this algorithm is obviously $\Theta(n)$. Note that this algorithm uses $\Theta(n)$ space. Can you modify it so it uses $\Theta(1)$ space?

2 Dynamic Programming, an introduction

The FastFib algorithm is an example of dynamic programming. The idea behind dynamic programming is to avoid recomputation of partial results. It uses the familiar divide and conquer approach; in addition it uses some data structure (usually some kind of table) to keep track of the solutions to subproblems to avoid recomputation. The idea is to use it when a straightforward divide and conquer approach would use the same values over and over again. Typically it is well suited for optimization problems.

3 Matrix Multiplication

Given two matrices A of size $p \times q$, and B of size $q \times r$, the product $A \cdot B = C$ is a matrix of size $p \times r$ where $c_{ij} = \sum_{i=1}^{q} a_{ik} b_{kj}$. Note that C contains pr entries, and that to compute each entry requires q multiplications and $q-1$ additions. Thus, computing C takes $\Theta(pqr)$ operations.

4 Multiplying three or more matrices

Now consider the case where we have more than two matrices to multiply. Suppose, for example, that we want to multiply four matrices, A of size 13×5, B of size 5×89, C of size 89×3, and D of size 3×34. We want to compute the matrix $ABCD$ that has size 13×34.

There are five ways we can do the calculation of ABCD:

1. $((AB)C)D$. We compute the number of operations that this would require. The product $AB = X$ requires $13 \cdot 5 \cdot 89 = 5,785$ multiplications to produce a matrix of size 13×89. The product $XC = Y$ requires $13 \cdot 89 \cdot 3 = 3,471$ multiplications to produce a matrix of size 13×3. The product YD requires $13 \cdot 3 \cdot 34 = 1,326$ multiplications to produce the final matrix. In total, this is $10,582$ multiplications.

2. $(AB)(CD)$ requires 54,201 multiplications.
3. $(A(BC))D$ requires 2,856 multiplications.
4. $A((BC)D)$ requires 4,055 multiplications.
5. $A(B(CD))$ requires 26,418 multiplications.

The best parenthesization is nearly 20 times better than the worst one! It thus pays to think about how to multiply matrices before you actually do it. Let us now formalize the problem.

5 Matrix Chain Multiplication (MCM)

Input: A sequence $A_1, A_2, ..., A_n$ of matrices, of size $p_0 \times p_1$, $p_1 \times p_2$, $p_2 \times p_3$, ..., $p_{n-1} \times p_n$, respectively.

Output: The smallest number of multiplications/operations to find the product, and the order in which the matrices should be multiplied.

Let us take a divide-and-conquer approach to the problem. To do that we define $m[i,j]$ to be the smallest number of multiplications necessary to find the subproduct $A_i \cdot A_{i+1} \cdot ... \cdot A_j$. Suppose that the best place to "split" the sequence is at k. Then the computation will look like:

$$(A_i \cdot ... \cdot A_k) \cdot (A_{k+1} \cdot ... \cdot A_j) \tag{1}$$

Note that $m[i,k]$ is the number of multiplications needed for $(A_i \cdot ... \cdot A_k)$, and $m[_{k+1}, j]$ is the number of multiplications needed for $(A_{k+1} \cdot ... \cdot A_j)$. Note also that $(A_i \cdot ... \cdot A_k)$ is a matrix of size $p_{i-1} \times p_k$, that $(A_{k+1} \cdot ... \cdot A_j)$ is a matrix of size $p_k \times p_j$ and that their product uses $p_{i-1} \times p_k \times p_j$ multiplications.

We thus obtain the following recurrence, for $i < j$:

$$m[i,j] = \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

and if $i = j$, then we only have a single matrix, so there is no multiplying to do and $m[i,j] = 0$.

These observations produce the following algorithm:

```
MCM(p,i,j)
if i = j
  return 0
else
  min = infinity
  for k = i to j-1
    test = MCM(p,i,k) + MCM(p,k+1,j) + p[i-1] p[k] p[j]
    if test < min
      min = test
  return min
```

We will get the answer to the general problem if we start with `MCM(p,1,n)`.

Let us analyze the running time of this algorithm on input of size $n = j - i + 1$. First note that in each iteration of the loop `for k = i to j-1`, we have to compute `MCM(p,i,k)` and `MCM(p,k+1,j)`. This means that we are evaluating the following recursive calls during the call to `MCM(p,i,j)`:

```
MCM[i,i], MCM[i,i+1], ..., MCM[i,j-1]
```

and

```
MCM[i+1,j], MCM[i+2,j], ..., MCM[j,j]
```

Note that we make a total of $2n$ recursive calls. Note, furthermore, that each one of `MCM[i,i+1], ..., MCM[i,j-1]` will again make the recursive call `MCM[i,i]`. So the same recursive calls are made over and over and the same $m[i, j]$'s are being recomputed so many times that the algorithm is exponential time as we have written it.

We can speed up the computation if, as with the Fibonacci numbers, we work bottom-up instead of top-down and avoid the recomputation of the entries $m[i, j]$. To do this we will construct a table of values. The entry in position (i, j) will represent $m[i, j]$. Since we are interested in computing the product of four matrices we will need a table of size 4×4:

$$\begin{pmatrix} 0 & & & \\ X & 0 & & \\ X & X & 0 & \\ X & X & X & 0 \end{pmatrix}$$

Observations:

- We cannot have $i > j$ so the table below the main diagonal will be ignored.
- All the values on the main diagonal are 0. It corresponds to the case $i = j$, which means we only have a single matrix and no work to do.

In order to do the computation bottom-up, we will compute each diagonal in turn, using the values that already exist in the table. The recurrence will tell us how to compute the diagonal values.

Example 1 *Computing $ABCD$, where A has size 13×5, B has size 5×89, C has size 89×3 and D has size 3×34. This means that $p_0 = 13$, $p_1 = 5$, $p_2 = 89$, $p_3 = 3$ and $p_4 = 34$. First we do the computation along the diagonal just above the main diagonal:*

$$\begin{array}{lcl cl cl} m[1,2] &=& p_0p_1p_2 &=& 13 \cdot 5 \cdot 89 &=& 5,785 \\ m[2,3] &=& p_1p_2p_3 &=& 5 \cdot 89 \cdot 3 &=& 1,335 \\ m[3,4] &=& p_2p_3p_4 &=& 89 \cdot 3 \cdot 34 &=& 9,078 \end{array}$$

Now do the computation along the next diagonal:

$$\begin{array}{lcl} m[1,3] &=& \min\{m[1,1] + m[2,3] + p_0p_1p_3, m[1,2] + m[3,3] + p_0p_2p_3\} \\ &=& \min\{1530, 9256\} \\ &=& 1530 \\ m[2,4] &=& \min\{m[2,2] + m[3,4] + p_1p_2p_4, m[2,3] + m[4,4] + p_1p_3p_4\} \\ &=& \min\{24208, 1845\} \\ &=& 1845 \end{array}$$

and finally, we compute the last entry, the one we're really looking for:

$$\begin{array}{lcl} m[1,4] &=& \min\{m[1,1] + m[2,4] + p_0p_1p_4, \\ && m[1,2] + m[3,4] + p_0p_2p_4, m[1,3] + m[4,4] + p_0p_3p_4\} \\ &=& \min\{4055, 54201, 2856\} \\ &=& 2856 \end{array}$$

The final table is:

$$\begin{pmatrix} 0 & 5,785 & 1,530 & 2,856 \\ X & 0 & 1,335 & 1,845 \\ X & X & 0 & 9,708 \\ X & X & X & 0 \end{pmatrix}$$

6 Dynamic Programming, in more depth

In order to apply a dynamic programming approach to solve a problem, the problem must have two properties:

Optimal substructure: the optimal solution to a problem contains, within it, the optimal solutions to its subproblems;

Overlapping subproblems: the total numberof subproblems is small and the obvious recursive algorithm solves the same subproblems over and over.

The matrix chain multiplication example has both these properties.

In order to solve a dynamic programming problem, we must perform the following steps:

1. Describe the structure of the optimal solutions. For example: In order to minimize the multiplications for $A_i ... A_j$ we have to minimize the total number of multiplications needed for computing the matrix $X = A_i ... A_k$ and the matrix $Y = A_{k+1} ... A_j$ for some k and for computing the product of X and Y.

2. Define the solutions recursively. For example, for $i < j$, we define $m[i,j] = \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$ if $i < j$.

3. Compute the values bottom-up. For example, the table of values all $m[i,j]$s.

4. Construct an optimal solution from the computed values. For example, in addition to the number of multiplications in the optimum parenthesization, we would like to know what the optimum parenthesization is. We show below how to do this.

The dynamic programming algorithm for matrix chain multiplication is:

```
MCM(p,n)

// initialize the main diagonal
for i = 1 to n
  m[i,i] = 0

// for each diagonal
for d = 2 to n
  // for each entry on the diagonal
  for i = 1 to n-d+1
    j = i+d-1
    m[i,j] = infinity
    // check all possible k
    for k = i to j-1
      q = m[i,k] + m[k+1,j] + p[i-1] p[k] p[j]
      if q < m[i,j] then
        m[i,j] = q
        s[i,j] = k
return m[1,n]
```

The running time of this iterative algorithm is $\Theta(n^3)$. The values $s[i,j]$ are kept in order to construct the optimal prenthesization as follows: for each $m[i,j]$, use $k = s[i,j]$ to discover that $m[i,j]$ was produced using $m[i,k]$ and $m[k+1,j]$.

7 Memoization

The version of matrix chain multiplication that we wrote was iterative. In re-writing the algorithm using dynamic programming, we changed the form of the algorithm from recursive to iterative. Instead we can use a table to store the values, and at the beginning of each recursive call check to see if the value we want already exists. If it does, then we re-use it. Otherwise we compute it. This technique is called memoization. It allows you to keep the top-down structure of the algorithm, but look up, instead of re-compute, the smaller values.

Consider writing a memoized version of the Fibonacci numbers solution:

- The first time we compute F_i we will store in into an array in position i.
- We will check the array before doing any recursion. If the value is there, we will use it. Otherwise we will compute the answer recursively.

This allows us to maintain the familiar structure of the problem but without the exponential costs of re-computing all the values. The memoized algorithm for Fibonacci numbers is:

```
Memo-Fib(n)
for i = 0 to n
  F[i] = 0
return Find-Fib(n)

Find-Fib(n)
if F[n] > 0 then
  return F[n]
else
  if n = 0 or n = 1 then
    F[n] = 1
    return 1
  else
    F[n] = Find-Fib(n-1) + FindFib(n-2)
    return F[n]
```

8 The $0-1$ knapsack problem

Suppose a thief finds n items in a safe. Each item has a weight and a value. Let the set of items be given by $S = \{1, 2, ..., n\}$ and let $w_1, ..., w_n$ be the corresponding weights and $v_1, ..., v_n$ be the corresponding values of the n items. The thief wants to take as valuable a load as possible, but she can only carry so many pounds, say C, in her knapsack.

Here is our first strategy: keep choosing the item with highest value/weight ratio that can fit the knapsack until no remaining item can fit. This is what is called a greedy strategy!

Example 2 *Let the size of the knapsack be 50 lbs. Suppose there are* 3 *items:*

1. $w_1 = 10lbs$, $v_1 = \$60$, *value/weight* $= \$6/lb$.
2. $w_2 = 20lbs$, $v_2 = \$100$, *value/weight* $= \$5/lb$.
3. $w_3 = 30lbs$, $v_3 = \$120$, *value/weight* $= \$4/lb$.

Our strategy tells us to take the first two items for a total values of \$160. *A better plan, however, is to take items 2 and 3 with total value* \$220.

So our greedy strategy fails miserably. Let us develop a dynamic programming solution to the $0-1$ knapsack problem.

8.1 Optimal substructure

First we have to check that the problem has the optimal substructure property. Let K be a most valuable subset of items (so $K \subset S$). If item n belongs to K then $K - \{n\}$ must be a most valuable load for the subproblem defined by items $S - \{n\}$ and knapsack weight $C - w_n$. If item $n \notin K$ then K must be a most valuable load for the subproblem defined by items $S - \{n\}$ and knapsack weight C. So the problem does have the optimal substructure property.

8.2 Recursive solution

Let $T[i, c]$ be the optimal value of a 0-1 knapsack problem with i items and knapsack weight c. Then $T[i, c]$ is defined recursively as follows:

$$T[i, c] = \max\{T[i-1, c], T[i-1, c-w_i] + v_i\}.$$

(If $c - w_i < 0$ then $T[i, c] = T[i-1, c]$ – this is the case when the last item, item i of weight w_i, is too big.) Note that the subproblems overlap. So, we can use dynamic programming. Note that to obtain $T[i, c - w_i]$, we also need to compute $T[i-1, c-w_i]$. So, as in the Fibonacci example, the recursive solution solves the same problem again and again, and we should use dynamic programming to compute $T(n, C)$.

8.3 Bottom-up, dynamic programming algorithm

Note that to compute entry $T[i, c]$ of the table $T[1..n, 1..C]$, we need to look up the entries $T[i-1, c]$ and $T[i-1, c-w[i]$. So, assuming the rows are numbered top to bottom and columns are numbered from left to right, we need to to fill the table T top to bottom and left to right.

```
Knapsack(C,v,w,n)
// C is the knapsack weight, v[1..n] is the array of item values and w[1..n] is
// the array of item weights; let T[0..n, 0..C] be the table of solutions to
subproblems.
for i = 0 to n
  T[i, 0] = 0
for c = 0 to C
  T[0, c] = 0
for c = 1 to C
  for i = 1 to n
    if c-w[i] < 0 or T[i-1,c] > T[i-1,c-w[i]] + v[i] then
      T[i,c] = T[i-1,c]
    else
      T[i,c] = T[i-1, c-w[i]] + v[i]
return T[n,C]
```

The running time is $\Theta(nC) = \Theta(n2^{\log C})$.

Example 3 *Let us solve the 0-1 knapsack problem with inputs* $(v_1, v_2, v_3) = (10, 15, 20)$, $(w_1, w_2, w_3) = (1, 2, 3)$ *and* $C = 5$. *We show our work in the* 4×6 *table* T *obtained by the algorithm.*

		0	1	2	3	4	5
	0	0	0	0	0	0	0
6	1	0	10	10	10	10	10
10	2	0	10	15	25	25	25
20	3	0	10	15	25	30	35

and the optimal solution contains items 2 *and* 3 *and has value* 35.

We have the value of the optimal solution. Think about how would we find the optimal solution itself?

9 The Maximum Sum Subvector problem (MSS)

We consider problem 5 from homework 2. Given a vector X of N real numbers we would like to find the maximum sum of entries found in any contiguous subvector of the input. For instance if the input vector is

```
31   -41   59   26   -53   58   97   -93  -23   84
```

then the program returns the sum of entries in X[3..7], or 187. The problem is easy when all the entries are positive – the maximum subvector is the entire input vector. The rub comes when some of the numbers are negative: should we include a negative number in hopes that the positive numbers to its sides will compensate for its negative contribution? Of course, if the entries are all negative then zero should be returned.

The trivial algorithm that checks every possible interval runs in $\Theta(n^3)$ time. With a bit more thought, you could improve the trivial algorithm to run in $\Theta(n^2)$ time. For your homework, you designed an even faster, $\Theta(n \log n)$ divide and conquer algorithm for this problem. We will now describe an optimal algorithm that runs in $\Theta(n)$ time and uses dynamic programming. Let us develop our algorithm:

1. We first need to describe the structure of the optimal solutions. We note that the maximum sum subvector of $X[1..j]$ is either the maximum sum subvector in $X[1..j-1]$ or it is a subvector that ends in position j.

2. We define the solutions recursively. For $j > 1$, let s_j be the sum of the entries in the maximum sum subvector in $X[1..j]$ and let m_j be the sum of the entries in the maximum sum subvector that ends in position j. Then

$$m_j = \max\{m_{j-1} + X[j], 0\},$$

$$s_j = \max\{s_{j-1}, m_j\}.$$

3. We compute the values bottom-up as described in the following algorithm.

```
MSS(X,N)
m = 0
s = 0
for j = 1 to n
  m = max{m+X[j], 0}
  s = max{s, m}
return s
```

Note that we do not use a table to store all $m_1, m_2, ...$ and $s_1, s_2, ...$. This is because, in iteration j, only values m_{j-1} and s_{j-1} are used.

4. We construct an optimal solution from the computed values. Note that the above algorithm does not return the subvector Z whose sum of entries is maximum – it just returns the sum of the entries. For homework, do modify this algorithm so it also returns the first and last position of the optimal subvector.

10 Longest Common Subsequence (LCS)

Consider two sequences of characters $X = x_1 x_2 ... x_m$ and $Z = z_1 z_2 ... z_k$. Z is a **subsequence** of X if there is an ascending sequence $i_1, i_2, ..., i_k$ such that for all j, $x_{i_j} = z_j$. For example, let $X = ABCAB$ and $Z = ABB$; then $i_1 = 1, i_2 = 2$ and $i_3 = 5$. A **common subsequence** for two sequences X_1 and X_2 is a subsequence of both X_1 and X_2. For example, if $X_1 = ababca$ and $X_2 = aabb$ then bb, abb, and aab are (some of the) subsequences common to X_1 and X_2.

We define the Longest Common Subsequence (LCS) problem as follows:

Input: Two sequences $X = x_1 ... x_m$ and $Y = y_1 ... y_n$.

Output: A longest sequence $Z = z_1 ... z_k$ such that Z is a subsequence common to both X and Y.

We will use dynamic programming to solve this problem.

10.1 Optimal substructure

First we have to check that the problem has the optimal substructure property. Let $X = x_1 ... x_m$ and $Y = y_1 ... y_n$ be two sequences and let $Z = z_1 ... z_k$

be a longest common subsequence for X and Y. We observe:

1. If $x_m = y_n$ then $z_k = x_m = y_n$ and $z_1 ... z_{k-1}$ must be a LCS of $x_1 ... x_{m-1}$ and $y1 ... y_{n-1}$.
2. If $x_m \neq y_n$ then two cases are possible:
    - If $z_k \neq x_m$ then $z_1 ... z_k$ is a LCS of $x_1 ... x_{m-1}$ and $y_1 ... y_n$. (We can throw out x_m and not affect the result).
    - If $z_k \neq y_n$ then $z_1 ... z_k$ is a LCS of $x_1 ... x_m$ and $y_1 ... y_{n-1}$. (We can throw out y_n and not affect the result).

Note that one of the two cases must hold since z_k cannot be equal to both x_m and y_n. Note also that both cases could hold.

10.2 Recursive solution

Let $c[i,j]$ be the length of a LCS of $x_1 ... x_i$ and $y_1 ... y_j$. Then

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } x_i \neq y_j \end{cases}$$

Note that the subproblems overlap. So, we can use dynamic programming.

10.3 Bottom-up computation

As with the matrix chain multiplication problem we will keep a table of values and compute the table entries using the recursive solution above. The table will have m rows and n columns. The rows will represent the sequence X and the columns will represent the sequence Y. Our goal will be to compute $c[m,n]$ since that gives us the LCS of X and Y. We start the computation as follows:

$$\begin{aligned} c[i,0] &= 0 \text{ for every } i \\ c[0,j] &= 0 \text{ for every } j \end{aligned}$$

We can compute the remaining table entries by observing that $c[i,j]$ will be either:

$$c[i-1,j-1]+1 \text{ or } \max\{c[i-1,j],c[i,j-1]\}.$$

Note that $c[i,j]$ is defined in terms of entries that are either above and/or to the left of it, so we should be filling the table row by row, from top to bottom and from left to right

Example 4 *Let $X = ABC$ and $Y = BCDE$. We start with the following table:*

$$\begin{array}{cccccccc}
 & & \epsilon & B & C & D & E \\
 & & 0 & 1 & 2 & 3 & 4 \\
\epsilon & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 1 & 0 & & & & \\
B & 2 & 0 & & & & \\
C & 3 & 0 & & & &
\end{array}$$

We compute the first row and obtain $c[1,j]=0$ for $1 \le j \le 4$ since $x_1 \neq y_j$ for $1 \le j \le 4$ and $\max\{c[0,j],c[1,j-1]\} = \max\{0,0\} = 0$:

$$\begin{array}{cccccccc}
 & & \epsilon & B & C & D & E \\
 & & 0 & 1 & 2 & 3 & 4 \\
\epsilon & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 1 & 0 & 0 & 0 & 0 & 0 \\
B & 2 & 0 & & & & \\
C & 3 & 0 & & & &
\end{array}$$

Computing the second row:

- $c[2,1] = c[1,0]+1 = 1$ *since* $x_2 = y_1$.
- $c[2,2] = \max\{c[1,2],c[2,1]\} = \max\{0,1\} = 1$ *since* $x_2 \neq y_2$.
- $c[2,3] = \max\{c[1,3],c[2,2]\} = \max\{0,1\} = 1$ *since* $x_2 \neq y_3$.
- $c[2,4] = \max\{c[1,4],c[2,3]\} = \max\{0,1\} = 1$ *since* $x_2 \neq y_4$.

to obtain:

$$\begin{array}{ccccccc}
 & & \epsilon & B & C & D & E \\
 & & 0 & 1 & 2 & 3 & 4 \\
\epsilon & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 1 & 0 & 0 & 0 & 0 & 0 \\
B & 2 & 0 & 1 & 1 & 1 & 1 \\
C & 3 & 0 & & & &
\end{array}$$

Computing the third row:

- $c[3,1] = \max\{c[2,1], c[3,0]\} = \max\{1,0\} = 1$ *since* $x_3 \neq y_1$.
- $c[3,2] = c[2,1] + 1 = 2$ *since* $x_3 = y_2$.
- $c[3,3] = \max\{c[2,3], c[3,2]\} = \max\{1,2\} = 2$ *since* $x_3 \neq y_3$.
- $c[3,4] = \max\{c[2,4], c[3,3]\} = \max\{1,2\} = 2$ *since* $x_3 \not y_4$.

This completes the table:

$$\begin{array}{ccccccc}
 & & \epsilon & B & C & D & E \\
 & & 0 & 1 & 2 & 3 & 4 \\
\epsilon & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 1 & 0 & 0 & 0 & 0 & 0 \\
B & 2 & 0 & 1 & 1 & 1 & 1 \\
C & 3 & 0 & 1 & 2 & 2 & 2
\end{array}$$

and the longest common subsequence has length $c[3,4] = 2$.

Here is a formal description of the algorithm:

```
LCSLength(X,Y)

m = length(X), n = length(Y)

for i = 0 to m
  c[i,0] = 0
for j = 0 to n
  c[0,j] = 0
```

```
for i = 1 to m
  for j = 1 to n
    if X[i] = Y[j] then
      c[i,j] = 1 + c[i-1,j-1]
    else
      c[i,j] = max{c[i-1,j], c[i,j-1]}

return c[m,n]
```

Let us analyze the running time. The initializations of the first row and the first column each take $\Theta(m)$ and $\Theta(n)$ steps, respectively. The nested for-loops require $\Theta(mn)$ steps. So, the overall algorithm takes $\Theta(mn)$ steps.

10.4 Constructing the optimal solution

In order to construct the optimal solution we have to save a pointer for each $c[i,j]$ that will indicate us which one of $c[i-1,j]$, $c[i-1,j-1]$ or $c[i,j-1]$ was used to produce $c[i,j]$. There are three possibilities:

1. $c[i,j] = c[i-1,j-1]+1$. In this case we need a pointer diagonally to $c[i-1,j-1]$.
2. $c[i,j] = c[i-1,j]$ because $c[i-1,j] \leq c[i,j-1]$. In this case we need a pointer up to $c[i-1,j]$.
3. $c[i,j] = c[i,j-1]$. In this case we need a pointer back to $c[i,j-1]$.

```
LCSSaveOpt(X,Y)

m = length(X), n = length(Y)

for i = 0 to m
  c[i,0] = 0
for j = 0 to n
  c[0,j] = 0

for i = 1 to m
  for j = 1 to n
```

```
    if X[i] = Y[j] then
      c[i,j] = 1 + c[i-1,j-1]
      p[i,j] = "\"
    else if c[i-1,j] >= c[i,j-1] then
      c[i,j] = c[i-1,j]
      p[i,j] = "|"
    else
      c[i,j] = c[i,j-1]
      p[i,j] = "--"

return c[m,n]
```

Example 5 *The $p[1..3, 1..4]$ table for $X = ABC$ and $Y = BCDE$ looks as follows:*

		ϵ	B	C	D	E
			1	2	3	4
ϵ	0					
A	1		|	|	|	|
B	2			$--$	$--$	$--$
C	3		|		$--$	$--$

Now, constructing the optimal string is very similar to finding the optimal parenthesization for the matrix chain multiplication. You simply start in position $p[3, 4]$ and follow the ”direction signals” back, adding characters to the LCS when a ”s̈ignal is encountered. You stop when you reach either boundary.