# Dynamic Programming

## Chapter 8

2016

# What is Dynamic Programming ?

- Dynamic programming is a general problem solving method that was developed in the field of mathematical programming. This field is concerned with solving mathematical optimization problems.

- Dynamic programming was invented by the American mathematician Richard Bellman who published a book in 1957 on this subject.

- Dynamic programming has similarities with divide and conquer, because it solves problems by combining solutions to sub-problems. However, there is one important difference: problems share sub-problems, and dynamic programming does not solve them repeatedly.

- Dynamic programming is usually applied to optimization problems. In such problems there can be many possible solutions and we are required to find a solution of optimal cost – an optimal solution.

# Development a Dynamic Programming Algorithm

- The development of a dynamic programming algorithm follows 4 steps:
  - Characterize the structure of an optimal solution
  - Recursively define the cost of an optimal solution
  - Compute the cost of an optimal solution in a bottom-up fashion
  - Compute an optimal solution from the computed information

# Matrix Chain Multiplication

- Consider to matrices $A(m,n)$ and $B(n,p)$.
- The computation of the product $C(m,p) = A \times B$ requires $m \times n \times p$ scalar multiplications. WHY?

- Matrix multiplication is associative, i.e. $(A \times B) \times C = A \times (B \times C)$.
- This means that the multiplication of a series of matrices can be done in several ways, using different parenthesizations.

- When matrices have different dimensions we need to be concerned about their *parenthesization* to compute the matrix chain product, as different parenthesizations can incur very different numbers of scalar multiplications !

# Matrix Chain Multiplication - Motivation

- Consider the multiplication of a chain of three matrices: $A_1(10,100)$, $A_2(100,5)$ and $A_3(5,50)$.
- If we multiply according to the parenthesization $((A_1 \times A_2) \times A_3)$ then:
  - We need $10 \times 100 \times 5 = 5000$ scalar multiplications in computing the product $A_{12}(10,5) = A_1 \times A_2$
  - Plus $10 \times 5 \times 50 = 2500$ scalar multiplications to compute the product $A_{123}(10,50) = A_{12} \times A_3$
  - Totaling 7500 scalar multiplications
- If we multiply according to the parnthesization $(A_1 \times (A_2 \times A_3))$ then:
  - We need $100 \times 5 \times 50 = 25000$ scalar multiplications in computing the product $A_{23}(100,50) = A_2 \times A_3$
  - Plus $10 \times 100 \times 50 = 50000$ scalar multiplications to compute the product $A_{123}\ (10,50) = A_1 \times A_{23}$
  - Totaling 75000 scalar multiplications
- Note that the first parenthesization is more efficient by a factor of 10 !

# Matrix Chain Multiplication – The Problem

- Given a sequence or chain $A_1, A_2, \ldots, A_n$ of $n$ matrices we seek to compute an optimal parenthesization for computing the product $A_1 \times A_2 \times \ldots \times A_n$.
- Because matrix multiplication is associative we can compute the product in several ways according to all its possible full parenthesizations. For example, if $n=4$ the product $A_1 \times A_2 \times A_3 \times A_4$ can be computed in any of five ways:

  $(A_1 \times (A_2 \times (A_3 \times A_4)))$
  $(A_1 \times ((A_2 \times A_3) \times A_4))$
  $((A_1 \times A_2) \times (A_3 \times A_4))$
  $((A_1 \times (A_2 \times A_3)) \times A_4)$
  $(A_1 \times (A_2 \times (A_3 \times A_4)))$
- Given that the matrix $A_i$ has dimension $(p_{i-1}, p_i)$ for all $1 \leq i \leq n$, the problem asks for finding a full parenthesization that minimizes the total number of scalar multiplications.

# Counting full parenthesizations

- Let $P(n)$ the number of full parenthesizations of the product of $n$ matrices. If we split between $A_k$ and $A_{k+1}$ we obtain two sub-sequences that can parenthesized independently. $k$ takes values between 1 and $n$–1, so summing for all $k$-s we obtain the following recurrence:

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n \geq 2 \end{cases}$$

- Solving this recurrence we obtain that $P(n) = C(n–1)$ where:

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

  is the *Catalan number* studied by combinatorics. Prove it as homework.

- It can be shown that Catalan number grows exponentially with $n$. Prove this as homework, as well. This means that generating all the full parenthesizations and selecting the optimal one is not acceptable !
- We need to look for an alternate strategy for determining the optimal parenthesization of a matrix chain.

# Solving the recurrence for $P(n)$

- Let $f(x)$ or $f$ be the generator function of sequence $P(n) = p_n$.

$$
\begin{aligned}
f(x) &= \sum_{n=0}^{\infty} p_i x^n,\ p_1 = 1 \\
f^2(x) &= (\sum_{n=0}^{\infty} p_i x^n)^2 = p_0^2 + 2p_0 x + \sum_{ngeq2} ((\sum_{k=0}^{n-1} p_k p_{n-k}) + p_0 p_n + p_n p_0) x^n \\
&= p_0^2 + 2p_0 x + \sum_{n\geq 2} (p_n + 2p_0 p_n) x^n = \\
&= p_0^2 + 2p_0 x + \sum_{n\geq 2} p_n x^n + 2p_0 \sum_{n\geq 2} p_n x^n \\
&= p_0^2 + 2p_0 x + f(x) - p_0 - x + 2p_0 (f(x) - p_0 - x) \\
f^2 &= -p0^2 - p_0 - x + f(2p_0 + 1) \\
& f^2 - f(2p_0 + 1) + p_0^2 + p_0 + x = 0 \\
\Delta &= (2p_0 + 1)^2 - 4(p_0^2 + p_0 + x) = 1 - 4x \geq 0 \text{ dnd } x \leq 1/4 \\
f(x) &= p_0 + 1/2 \pm 1/2\sqrt{1 - 4x}
\end{aligned}
$$

- The correct solution is with "-" as $f(x)$ is increasing. It follows:

$$f(x) = p_0 + 1/2(1 - \sqrt{1 - 4x})$$

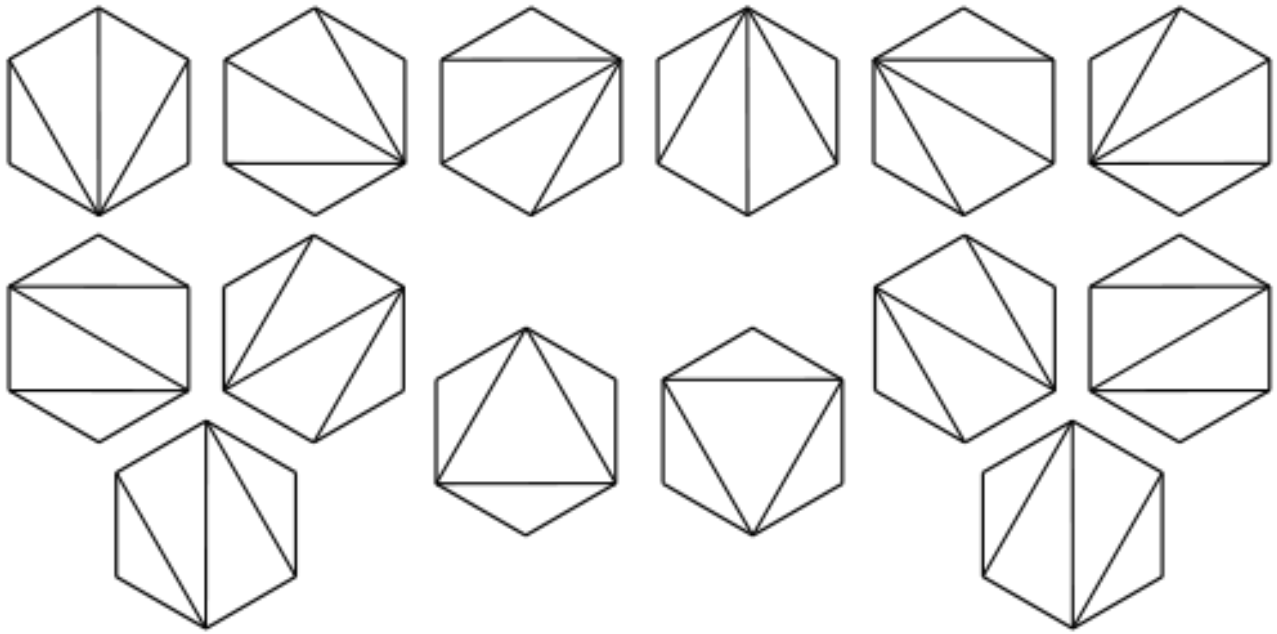
# Solving the recurrence for *P*(*n*)

- We write *f*(*x*) as a Taylor series around *x* = 0, starting with:

$$
\begin{aligned}
g(x) &= \sqrt{1-4x} \\
g(x) &= g(0) + xg^{(1)}(0)/1! + xg^{(2)}(0)/2! + \cdots = 1 + \sum_{n\le 1} \alpha_n x^n \\
g^{(1)}(x) &= -2(1-4x)^{-1/2} \\
g^{(2)}(x) &= -4(1-4x)^{-3/2} \\
g^{(3)}(x) &= -8\times 3(1-4x)^{-5/2} \\
g^{(4)}(x) &= -16\times 3\times 5(1-4x)^{-7/2} \\
g^{(5)}(x) &= -32\times 3\times 5\times 7(1-4x)^{-9/2} \\
&\cdots \\
g^{(n)}(x) &= -2^n\times 1\times 3\times\cdots\times(2n-3)(1-4x)^{-(2n-1)/2} \\
&= -2(2n-2)!/(n-1)!(1-4x)^{-(2n-1)/2} \\
g(0) &= = 1 \\
g^{(n)}(0) &= -2(2n-2)!/(n-1)! \\
\alpha_n &= -2\frac{(2n-2)!}{n!(n-1)!} = (-2)\frac{1}{n}\binom{2n-2}{n-1},\ \lim_{n\to\infty}\frac{\alpha_{n+1}}{\alpha_n} = 4,\ \text{i.e. convergenece radius is } \frac{1}{4} \\
p_n &= -\frac{1}{2}\alpha_n = \frac{1}{n}\binom{2n-2}{n-1} = C(n-1)\ \text{q.e.d.}
\end{aligned}
$$

# Catalan number

- Catalan number $C(n)$ is equal to the number of triangulations of a convex polygon with $n+2$ vertices into $n$ triangles.
- For example, if $n=4$ there are 14 triangulations.
- This shows that there are 14 parenthesizations of a product of 5 matrices.



# Characterize the structure of an optimal solution

- For matrix-chain multiplication, the first step is to examine the structure of optimal parenthesizations. We seek an optimal solution to an instance of the matrix-chain multiplication problem containing within it optimal solutions to sub-problem instances.
- Let $A_{i..j} = A_i \times A_{i+1} \times ... \times A_j$. An optimal parenthesization of $A_{i..j}$ splits the product between $A_k$ and $A_{k+1}$ for some $i \leq k < j$.
- The cost of an optimal parenthesization will be the cost of computing $A_{i..k}$ plus the cost of computing $A_{k+1..j}$ plus the cost of multiplying $A_{i..k}$ with $A_{k+1..j}$.
- The key point is that for the parenthesization of $A_{i..j}$ to be optimal it is necessary that the parenthesizations of $A_{i..k}$ and $A_{k+1..j}$ are optimal as well. This means that the matrix chain multiplication problem has the property of *optimal sub-structure*, i.e. an optimal solution of a problem contains within it optimal solutions for sub-problems.

# Recursively define the value of an optimal solution (I)

- For the current optimization problem, the second step is to examine a recursive solution. For the matrix-chain multiplication problem we choose as our sub-problems the problems of determining the minimum cost of a parenthesization of $A_i \times A_{i+1} \times ... \times A_j$, $1 \le i \le j \le n$.
- Let $m_{ij}$ denote the minimum number of scalar multiplications needed to compute matrix $A_{i..j}$. The minimum cost for the full product will be $m_{1n}$.
- If $i = j$ or a matrix-chain of one element (matrix) then we have $A_{i..j} = A_i$ and no scalar multiplications are required, so $m_{ii} = 0$ for $1 \le i \le n$.
- If $i < j$ the optimal parenthesization splits the product $A_i \times A_{i+1} \times ... \times A_j$, between $A_k$ and $A_{k+1}$ where $i \le k < j$. $m_{ij}$ equals the sum of the minimum costs for computing sub-products $A_{i..k}$ and $A_{k+1..j}$ plus the cost of multiplying these two sub-products together, i.e. $p_{i-1} \times p_k \times p_j$. We obtain:

$$m_{ij} = m_{ik} + m_{k+1j} + p_{i-1} p_k p_j$$

# Recursively define the value of an optimal solution (II)

- We need to check all the possible values for $k$ to find the one that yields the minimal cost.
- So finally we obtain the recurrence:

$$m_{ij} = \begin{cases} 0 & i = j \\ \min_{i \le k < j} m_{ik} + m_{k+1j} + p_{i-1} p_k p_j & i < j \end{cases}$$

- To keep track of how to construct an optimal solution:
  - We define $s_{ij}$ to be the value of $k$ at which we can split the product $A_i \times A_{i+1} \times ... \times A_j$ to obtain the optimal parenthesization
  - That is $s_{ij} = k$ such that $m_{ij} = m_{ik} + m_{k+1j} + p_{i-1} \times p_k \times p_j$ is minimal.

# Compute the value of an optimal solution – why top-down is not a good idea?

- It is easy to write a top-down recursive algorithm to compute $m_{ij}$ using the recurrence, following the idea of divide-and-conquer. However, this algorithm will have an exponential time. Why ?
- Notice that the number of sub-problems is equal to the number of pairs $(i,j)$ such that $1 \leq i \leq j \leq n$. This number is $\Theta(n^2)$. Why ?
- This means that are a lot of overlapping sub-problems in the different branches of the recursion tree for this problem. This is a property that we seek when applying dynamic programming.

# Compute the value of an optimal solution – bottom up

- The best approach to compute the minimal cost, avoiding to repeatedly solve sharing sub-problems, is the bottom-up approach. The computation is done in increasing values of $l = j–i$, i.e. $l = 0, 1, 2, \ldots, n–1$.
- If the matrix $A_i$ has dimension $(p_{i–1},p_i)$ for all $1 \leq i \leq n$, then the algorithm:
  - Takes the sequence $p_0, p_1, \ldots, p_n$ of size $n+1$ as input
  - And generates the tables $m(n,n)$ and $s(n,n)$ as output such that $m_{ij}$ is the cost for $1 \leq i \leq j \leq n$ and $s_{ij}$ is the value of $k$ for $1 \leq i < j \leq n$.

# The bottom up algorithm

MATRIX-CHAIN-ORDER($p$)

1. $n \leftarrow length[p] - 1$
2. **for** $i \leftarrow 1, n$ **do**
3. $\quad m[i, i] \leftarrow 0$
4. **for** $l \leftarrow 1, n - 1$ **do**
5. $\quad$ **for** $i \leftarrow 1, n - l$ **do**
6. $\quad\quad j \leftarrow i + l$
7. $\quad\quad m[i, j] \leftarrow \infty$
8. $\quad\quad$ **for** $k \leftarrow i, j - 1$ **do**
9. $\quad\quad\quad q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
10. $\quad\quad\quad$ **if** $q < m[i, j]$ **then**
11. $\quad\quad\quad\quad m[i, j] \leftarrow q$
12. $\quad\quad\quad\quad s[i, j] \leftarrow k$
13. **return** $m,\ s$

The time complexity of this algorithm is $\Theta(n^3)$.

# Example

- Consider a sequence of matrices $A_1(10,5)$, $A_2(5,20)$, $A_3(20,10)$, $A_4(10,15)$. The input is $p_0 = 10$, $p_1 = 5$, $p_2 = 20$, $p_3 = 10$, $p_4 = 15$.
- The output matrices *m* and *s* are:

| ***m*** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| **1** | **0** | **1000** | **1500** | **2500** |
| **2** | **–** | **0** | **1000** | **1750** |
| **3** | **–** | **–** | **0** | **3000** |
| **4** | **–** | **–** | **–** | **0** |

| ***s*** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| **1** | **–** | **1** | **1** | **1** |
| **2** | **–** | **–** | **2** | **3** |
| **3** | **–** | **–** | **–** | **3** |
| **4** | **–** | **–** | **–** | **–** |

- The optimal parenthesization corresponding to matrix *s* is:
  $(A_1 \times ((A_2 \times A_3) \times A_4))$ and it costs 2500 scalar multiplications.

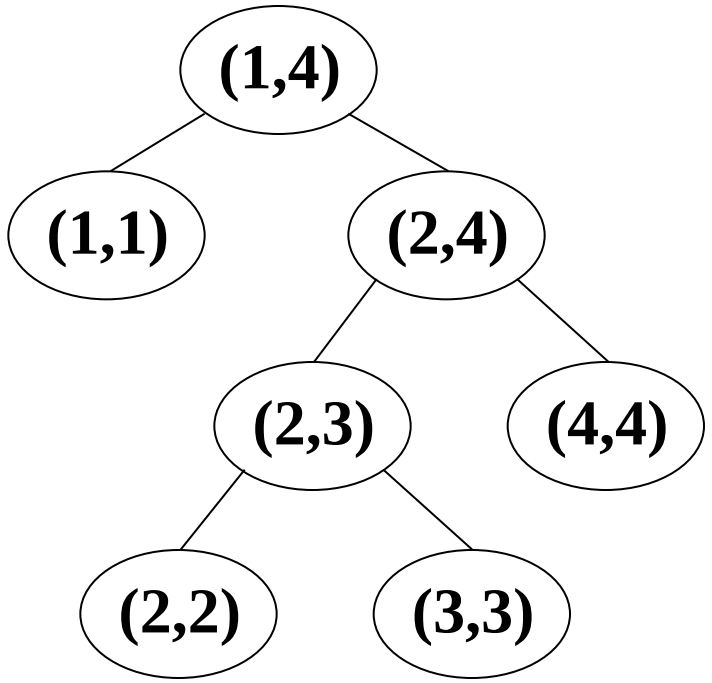
# Displaying the optimal parenthesization (I)

- The split points $s_{ij}$ define a tree structure. Each internal tree node is labeled with a triple $(i, j)$ with $1 \le i \le j \le n$.
- The root of the tree is $(1, n)$. If $s_{ij} = k$ then the root of left sub-tree of $(i,j)$ is $(i,k)$ and the right sub-tree is $(k+1,j)$.
- For the example shown on the previous slide the tree is:
- Displaying the optimal parenthesization can be done with an inorder traversal of this tree.

$s_{14} = 1$

$s_{24} = 3$

$s_{23} = 2$



# Displaying the optimal parenthesization (II)

```
PRINT-PAR(i, j)
1. if i = j then
2.     PRINT(i)
3. else
4.         PRINT('(')
5.         PRINT-PAR(i, s[i, j])
6.         PRINT('*')
7.         PRINT-PAR(s[i, j] + 1, j)
8.         PRINT(')')
```

Homework: Devise an algorithm that actually implements the matrix multiplication according to the multiplication pattern given by the optimal parenthesization.

# Implementation of matrix chain multiplication (I)

```
#include <stdio.h>
#define NMAX 100
#define INFTY 1e100

void matrix_chain_order(void);
void read_sizes(void);
void write_par(int i,int j);
void write_answer(void);

int p[NMAX];
double m[NMAX][NMAX];
int s[NMAX][NMAX];
int n;
```

```
void main() {
  read_sizes();
  while (n>0) {
    matrix_chain_order();
    write_answer();
    read_sizes();
  }
}
```

# Implementation of matrix chain multiplication (II)

```
void read_sizes() {
  int i;
  scanf("%d",&n);
  if (n>0) {
    for (i=0;i<=n;i++)
      scanf("%d",&p[i]);
  }
}
void write_par(int i,int j) {
  if (i==j)
    printf("%3d",i);
  else {
    printf("(");
    write_par(i,s[i][j]);
    printf("*");
    write_par(s[i][j]+1,j);
    printf(")");
  }
}
```

```
void write_answer() {
  int i,j;
  printf("Costs:\n");
  for (i=1;i<=n;i++) {
    for (j=1;j<=n;j++)
      printf("%9.0f",m[i][j]);
    printf("\n");
  }
  printf("Split points:\n");
  for (i=1;i<=n;i++) {
    for (j=1;j<=n;j++)
      printf("%3d",s[i][j]);
    printf("\n");
  }
  printf("Parenthesization:\n");
  write_par(1,n); printf("\n");
}
```

# Implementation of matrix chain multiplication (III)

```
void matrix_chain_order() {
  int i,j,k,l;

  for (i=1;i<=n;i++)
    m[i][i] = 0;
  for (l=1;l<n;l++) {
    for (i=1;i<=n-l;i++) {
      j = i+l;
      m[i][j] = INFTY;
      for (k=i;k<=j-1;k++) {
        double q = m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
        if (q<m[i][j]) { m[i][j] = q;s[i][j] = k; }
      }
    }
  }
}
```

# Text auto-formatting

- One feature of text editors is auto-formatting. An auto-formatting operation requires the automated division of text into lines of given maximum size such that lines have approximately equal lengths (in number of characters).
- A word is a sequence of consecutive characters that do not contain spaces. There must be one space between each two consecutive words.
- The sum of the squares of the left over spaces at the end of each line, excepting the last line, must be minimized to assure that the lines have approximately an equal number of characters.
- The input data consists a one or more data sets. A data set contains:
  - An integer N between 1 and 1000 representing the number of words
  - An integer K between 1 and 200 representing the maximum number of characters of each line.
  - The list of words separated by spaces. Each word has at most 50 characters
- The data sets are terminated with a 0.
- For each data set your program must print out the minimum cost.

# Method for solving text auto-formatting (I)

- Let $w_1$, ..., $w_n$ be the $n$ words and $|w|$ the length of word $w$. We impose $k \geq max\ \{|w_1|, ..., |w_n|\}$ (*). Otherwise at least one word will not fit into a line.
- Let $f(i)$ the cost function that must be minimized for the subsequence of words starting with word $i$. We must determine $f(1)$.
- Let $P(i)$ be set of all values $p \geq i$ such that subsequence $w_i$, ..., $w_p$ fits a single line, i.e. $|w_i| + ... + |w_p| + (p\text{-}i) \leq k$. According to (*), this set is always non-empty, as $i \in P(i)$. Let $p(i) = max\ P(i)$, so $P(i) = \{i, ..., p(i)\}$.
- If $p(i) = n$ then $f(i) = 0$, as the subsequence containing only word $w_n$ fits entirely on a single line, which is in fact the last line.

# Method for solving text auto-formatting (II)

- If $p(i) < n$ then there are left over words not assigned to lines. So, the minimum of $f(i)$ is obtained trying all the possibilities of „breaking” the subsequence in position $p$ (word $p$ staying on the current line, word $p$+1 moves to the next line), for all $p \in P(i)$, and choosing $p$ such that the following expression is minimal:

  $$f(i) = [k - (|w_i| + ... + |w_p| + (p\text{-}i))]^2 + f(p+1) \quad (**)$$

- Equation (**) allows to compute $f(i)$ for $i = n$ down to $i$=1.
- According to (*), we have the initialization $f(n) = 0$, as each word fits on a single line.
- Then we determine $p(n\text{-}1)$, $f(n\text{-}1)$, ... . The computation is well defined as at each step $i$, $f(i)$ depends only on $f(i+1)$, ...., $f(p(i))$, and these values were already determined in previous steps.

# Implementation of text auto formatting (I)

```
#include <stdio.h>
#include <string.h>
/* Lungimea maxima a unui cuvant */
#define CMAX 51
/* Numarul maxim de cuvinte */
#define NMAX 1001
/* Lungimea maxima a unei linii */
#define KMAX 201
/* Numarul de cuvinte dintr-un paragraf */
int n;
/* Lungimea maxima a unei linii */
int k;
/* Vectorul de cuvinte, numarat de la 1 */
char cuvinte[NMAX][CMAX];
/* Suma lungimilor cuvintelor wi, wi+1, ...wj */
int slcuv[NMAX][NMAX];
/* Costuri minime ale fiecarei subprobleme ce incepe cu un cuvant dat. */
int f[NMAX];
/* Costul minim */
long val;
```

# Implementation of text auto formatting (II)

```
/* Citeste datele: lungimea maxima a liniilor si lista de cuvinte */
void citeste_date() {
    int i;
    scanf("%d",&k);
    for (i=1;i<=n;i++) {
        scanf("%s",cuvinte[i]);
    }
}
/* Determina suma lungimilor cuvintelor i, i+1, ..., j in timp O(n^2) */
void sume_lungimi_cuvinte() {
    int i,l;
    for (i=1;i<=n;i++) {
        slcuv[i][i] = strlen(cuvinte[i]);
    }
    for (i=1;i<n;i++) {
        for (l=1;l<=n-i;l++) {
            int j = i+l;
            slcuv[i][j] = slcuv[i][j-1]+strlen(cuvinte[j]);
        }
    }
}
```

# Implementation of text auto formatting (III)

```
/* Determina costurile minime pentru o anumita lungime a liniei */
void determina_costuri_minime(int lun) {
    int k1,p;
    long W;
    for (k1=n;k1>=1;k1--) {
        if (slcuv[k1][n]+n-k1 <= lun) {
            /* Inseamna ca toate cuvintele de la k1 incolo,
               inclusiv k1, incap in ultima linie */
            f[k1] = 0;
        }
        else {
            /* Determinam unde trebuie rupta linia si costul aferent */
            p = k1;
            W = slcuv[k1][p]+p-k1-lun;
            f[k1] = LONG_MAX;
            while (W <= 0) {
                int t;
                t = W*W + f[p+1];
                if (t<f[k1]) { f[k1] = t; }
                p++;
                W = slcuv[k1][p]+p-k1-lun;
            }
        }
    }
}
```

# Implementation of text auto formatting (IV)

```
void proceseaza_date() {
    sume_lungimi_cuvinte();
    determina_costuri_minime(k);
    val = f[1];
}

int main() {
    scanf("%d",&n);
    while (n>0) {
        citeste_date();
        proceseaza_date();
        printf("%ld\n",val);
        scanf("%d",&n);
    }
    return 0;
}
```

# Arranging domino pieces

- Consider a sequence of *n* domino pieces. Each piece may be rotated with 180 degrees around its center. Compute the largest subsequence of consecutive pieces that can be obtained by applying rotations such that any two neighboring pieces have equal numbers on their neighboring halves. The dominos are represented as a matrix *a*[1..*n*][0..1].
- A solution is uniquely identified by the position of its starting element and its length.
- Example: if the dominos are (1 3) (4 5) (3 5) (2 3) (4 3) (1 4) then a solution is the subsequence starting with position 2 and length 3. The dominos of this solution are rotated as follows: (4 5) (5 3) (3 2). If the presence of rotation is encoded as 1 and the absence of rotation is encoded as 0, then the rotation pattern is: 0 1 1.
- Note that the problem space is exponential in size. Why ?

# Idea for the dominoes problem

- We denote with $c_{ij}$ the maximum length of a sub-sequence ending with element $i$, rotated or not as described by $j$, $1 \leq i \leq n$ and $0 \leq j \leq 1$.
- $c_{10} = c_{11} = 1$
- $c_{ij}$ can be computed recursively using dynamic programming:
  $c_{ij} = \max_{0 \leq k \leq 1} \alpha_{ijk}$ where
  if $a_{ij} \neq a_{i\text{-}11\text{-}k}$ then $\alpha_{ijk} = 1$ else $\alpha_{ijk} = 1 + c_{i\text{-}1k}$
- So we look if piece $i$ rotated as indicated by $j$ matches piece $i$-1 either rotated ($k$=1) or not ($k$=0). In case of a match, the maximum length is incremented, otherwise the length is reset to 1. We obviously take the maximum value from the two possibilities: piece $i$-1 rotated or not.
- Question: what information needs to be saved in order to be able to recover a solution ?

# Homework I

1. Let $X$ be a sequence of $n$ positive integers $x_1, x_2, \ldots, x_n$ and an integer $t$.
   - Devise an algorithm that checks if there is a sub-sequence of $X$ with the sum of its elements equal to $t$. For example, if $X = 2, 6, 3, 5, 4$ and $t = 5$, then the answer is *yes* because the sub-sequence 2, 3 fulfills the condition. However, if $X = 3,5,1$ and $t = 7$, then the answer is *no*.
   - Update your algorithm to print a solution, if the answer is *yes*.
   - Update your algorithm to compute the number of solutions and to print them.
2. Find the longest common sub-sequence of two sequences. For example the longest common sub-sequence of $X$ = 'A', 'B', 'C', 'B', 'D', 'A', 'B' and Y = 'B', 'D', 'C', 'A', 'B', 'A' is $X_2, X_5, X_6, X_7 = Y_1, Y_2, Y_4, Y_5$ = 'B', 'D', 'A', 'B'. See the textbook for the solution.

# Homework II

3. Given a list of $n$ coins, their values ($x_1$, $x_2$, ... , $x_n$), and the total sum $s$, find the minimum number of coins with sum equal to $s$ (we can use as many coins of one type as we want), or report that it is not possible to select coins in such a way that they sum up to $s$.
4. Let $a = (a_1, a_2, \ldots, a_n)$ be a sequence of $n$ integers. Deleting some elements, we obtain a subsequence of this sequence. The remaining subsequence is called increasing is its elements are in increasing order. For if our sequence is (7, 3, 8, 5, 13) then it contains the increasing subsequence (7, 8, 13), as well as the subsequence (7, 3, 5) which is not increasing. Determine the longest increasing subsequence of a given sequence.