ALGORITHMS AND DATA STRUCTURES I

Course 11

Dynamic Programming

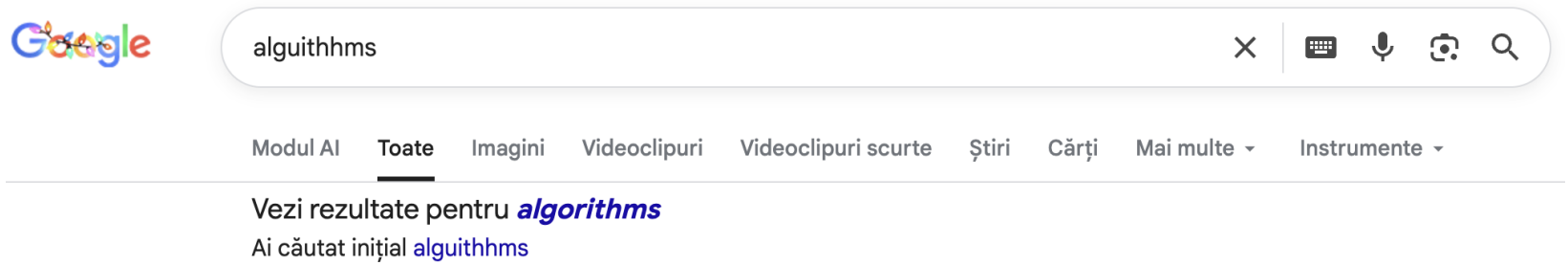
Previous Course

- Technique of locally optimal choice - "greedy algorithms"
  - Structure
  - The basic idea of the locally optimal choice technique
  - Examples
  - Verification of correctness and analysis of efficiency
  - Applications

Today Course

- What is dynamic programming?
- Main steps in applying dynamic programming
- Recurrence relationships: top-down vs. bottom-up development
- Applications
- Memoization technique

Motivation



How do you decide that I intended to write algorithms instead of alguithhms?

A measure of dissimilarity between words is evaluated

Motivation

What errors can occur when typing a word (text)

- Replacing a letter with another letter
  - alguithhms -> algoithhms
- Absence of a letter
  - algoithhms -> algorithhms
- Deleting an additional letter
  - algorithhms -> algorithms

How can the distance between two words be calculated?

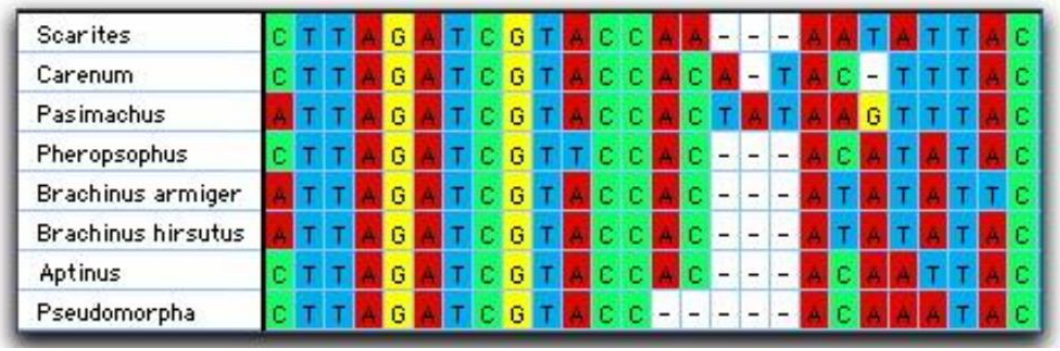
The minimum number of letter replacement/deletion/insertion operations that ensures the transformation of one of the words into the other

Example: alguithhms -> algoithhms -> algorithhms -> algoithms

replace insert delete

Motivation

Analysis of similarity between DNA sequences (DNA sequence alignment) which ensures the transformation of one of the words into the other



[www.sequence-alignment.com]

Note: Determining the match score between two DNA sequences is similar to determining the editing distance, with editing errors being replaced by mutations that cause: insertion/deletion/replacement of a nucleotide in either sequence

Editing distance calculation

Input: Consider two strings (words): x[1..m] and y[1..n]

Output: The minimum number of insertion, deletion, symbol replacement operations that allows the transformation of the string x[1..m] into the string y[1..n]

Idea: first analyze particular cases and then try to extend the solution to the general case

Notation: D[i,j] represents the editing distance between the partial strings (prefixes) x[1..i] and y[1..j]

Particular cases:

- i=0 (x is empty): D[0,j]=j
  - (j insertions are needed to transform x into y)

- j=0 (y is empty): D[i,0]=i
  - (i deletions are needed to transform x into y)

Editing distance calculation

Input: Consider two strings (words): x[1..m] and y[1..n]

Output: The minimum number of insertion, deletion, symbol replacement operations that allows the transformation of the string x[1..m] into the string y[1..n]

Notation: D[i,j] represents the editing distance between the partial strings (prefixes) x[1..i] and y[1..j]

Possible cases:

- If x[i]=y[j] then D[i,j] = D[i-1,j-1]
- If x[i]!=y[j] then three situations are analyzed:
  - x[i] is replaced by y[j]: D[i,j]=D[i-1,j-1]+1
  - x[i] is deleted: D[i,j]=D[i-1,j]+1
  - y[j] is inserted after x[i]: D[i,j]=D[i,j-1]+1

and the variant that leads to the smallest number of operations is chosen:

D[i,j]=min{D[i-1,j-1] , D[i-1,j], D[i,j-1]}+1

Editing distance calculation

Example: x= kitten y=sitting

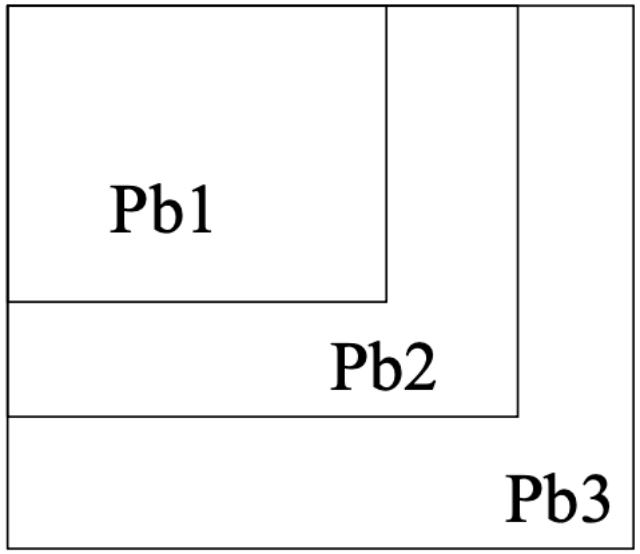
$$D[i,j] = \begin{cases} i & j = 0 \\ j & i = 0 \\ D[i-1, j-1] & x[i] = y[j] \\ \min\{D[i-1,j-1], D[i,j-1], D[i-1,j]\} + 1 & x[i] \neq y[j] \end{cases}$$

	""	s	i	t	t	i	n	g
""	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

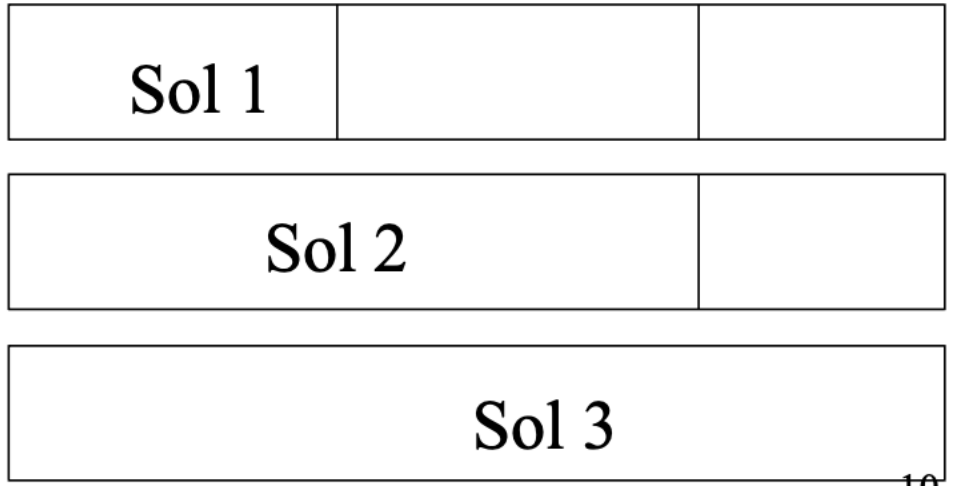
What is dynamic programming?

- It is a technique for designing algorithms for solving problems that can be decomposed into overlapping subproblems – it can be applied to optimization problems that have the property of optimal substructure (= an optimal solution to the problem is made up of optimal solutions to the subproblems)
- The peculiarity of the method lies in the fact that each subproblem is solved once and its solution is stored (in a table structure) so that it can be later used to solve the initial problem.

Decomposition of the problem into overlapping subproblems



The solution to problem Pb3 contains the solutions to (sub)problems Pb1 and Pb2



What is dynamic programming?

- Dynamic programming was developed by Richard Bellman in 1950 as a general method for optimizing decision processes.

- In dynamic programming, the word programming refers to planning and not to programming in the computer sense.

- The word dynamic refers to the manner in which tables are constructed in which retains information regarding partial solutions.

What is dynamic programming?

Dynamic programming is related to the division technique because it is based on dividing the initial problem into subproblems. However, there are some significant differences between the two approaches:

- division: the subproblems into which the initial problem is divided are independent, so the solution of a subproblem cannot be used in constructing the solution to another subproblem

- dynamic programming: subproblems are dependent (overlap) so that the solution of a subproblem is used in constructing the solutions of other subproblems (for this reason it is important that the solution of each solved subproblem be stored so that it can be reused) – problems for which the dynamic programming technique can be applied have the property of optimal substructure

What is dynamic programming?

The difference between decomposition into subproblems in the case of the division technique and dynamic programming

Problem decomposition in subproblems Solution building

Dynamic programming

Division technique

What is dynamic programming?

- Dynamic programming is also correlated with the locally optimal (greedy) choice strategy since both apply to optimization problems that have the property of optimal substructure and solutions are built incrementally.

- Unlike the greedy technique (which does not guarantee obtaining the optimal solution), dynamic programming allows obtaining an optimal solution.

Main steps in applying dynamic programming

1. Analyze the structure of the solution: determine how the solution of the problem depends on the solutions of the subproblems. This stage actually refers to checking the substructure property optimal and identifying the generic problem (the general form of the initial problem and each subproblem).

2. Identifying the recurrence relation that expresses the connection between the solution of the problem and the solutions of the subproblems. Usually, the recurrence relation involves the value of the optimality criterion.

3. Development of the recurrence relation. The relation is developed in ascending (TOP-DOWN) manner so as to build the table with the values associated with the subproblems.

4. The actual construction of the solution – is based on the information determined in the previous stage.

Developing recurrence relationships

There are two main approaches:

- Ascending - Bottom up: start from the particular case and generate new values based on the existing ones.

- Descending - Top down: the value to be calculated is expressed by previous values, which must be calculated in turn. This approach is usually implemented recursively (and most of the time leads to inefficient variants – efficiency can be achieved by the memoization technique

Developing recurrence relationships

Example 1. Calculating the m-th element of the Fibonacci sequence $f_1=f_2=1$; $f_n=f_{n-1}+f_{n-2}$ for n>2

Top-down approach

```
fib(m)
  IF (m=1) OR (m=2) THEN
    RETURN 1
  ELSE
    RETURN fib(m-1)+fib(m-2)
  ENDIF
```

Efficiency

$$T(m) = \begin{cases} 0 & m \leq 2 \\ T(m-1) + T(m-2) + 1 & m > 2 \end{cases}$$

T: 0 0 1 2 4 7 12 20 33 54

Fibonacii: 1 1 2 3 5 8 13 21 34 55

$$f_n \in O(\varphi^n), \varphi = (1+\sqrt{5})/2$$

T(m)=f_m-1 belong to $O(\varphi^m)$

Developing recurrence relationships

Example 1. Calculating the m-th element of the Fibonacci sequence $f_1=f_2=1$; $f_n=f_{n-1}+f_{n-2}$ for n>2

Bottom-up approach

```
fib(m)
  f[1]←1; f[2] ← 1;
 FOR i ← 3,m DO
    f[i] ← f[i-1]+f[i-2]
 ENDFOR
 RETURN f[m]
```

Efficiency

T(m)=m-2 => linear complexity

Remark:

- Time efficiency is paid by using additional space.
- The size of the additional space can be significantly reduced 2 additional variables are enough)

```
fib(m)
  f1←1; f2 ← 1;
 FOR i ← 3,m DO
    f2 ← f1+f2
    f1 ← f2-f1
 ENDFOR
 RETURN f[m]
```

Developing recurrence relationships

Example 2. Calculation of binomial coefficients C(n,k) (combinations of n taken by k)

$$C(n,k) = \begin{cases} 0 & n < k \\ 1 & k = 0 \; sau \; n = k \\ C(n-1,k) + C(n-1,k-1) & altfel \end{cases}$$

Top-down approach

```
comb(n,k)
  IF (k=0) OR (n=k) THEN
     RETURN 1
  ELSE
     RETURN comb(n-1,k)+comb(n-1,k-1)
ENDIF
```

Efficiency

Problem size: (n,k)

Dominant operation: addition

T(n,k)= 0 if k=0 or k=n

T(n,k)=T(n-1,k)+T(n-1,k-1), otherwise

No. of additions = no. of nodes in the recursive call tree

$$T(n,k) \geq 2^{\min\{k,n-k\}}$$

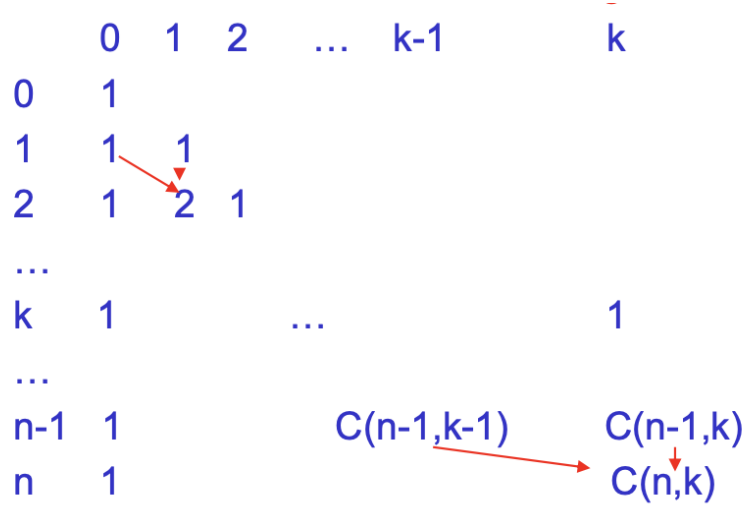
$$T(n,k) = \Omega(2^{\min\{k,n-k\}})$$

Developing recurrence relationships

Example 2. Calculation of binomial coefficients C(n,k) (combinations of n taken by k)

$$C(n,k) = \begin{cases} 0 & n < k \\ 1 & k = 0 \; sau \; n = k \\ C(n-1,k) + C(n-1,k-1) & otherwise \end{cases}$$

Top-down approach – Pascal triangle



$$d[i,j] = \begin{cases} d[i,0] = 1 & any \; i \\ d[i,i] = 1 & \\ d[i-1,j + d[i-1,j-1] & i < j \end{cases}$$

Developing recurrence relationships

```
Algorithm
Binomial_coefficients(n,k)
 FOR i←0,n DO
   FOR j ← 0,min{i,k} DO
    IF (j==0) OR (j==i) THEN
      d[i,j] ← 1
    ELSE
       d[i,j] ← d[i-1,j]+C[i-1,j-1]
    ENDIF
   ENDFOR
  ENDFOR
RETURN C[n,k]
```

Efficiency

Problem size: (n,k)

Dominant operation: addition

$T(n,k)=(1+2+...+k-1) +(k+...+k) =k(k-1)/2+k(n-k+1)$

$T(n,k) = \Theta(nk)$

Remark: If only C(n,k) needs to be calculated, it is sufficient to use an array with k elements as additional space.

Application of dynamic programming

Longest strictly increasing substring

Let $a_1, a_2, \ldots, a_n$ be a sequence. Determine the longest substring having the property $a_{j1} < a_{j2} < \ldots < a_{jk}$ (a strictly increasing substring with the maximum number of elements).

Problem constraint: $a_{j1} < a_{j2} < \ldots < a_{jk}$

Optimality criterion: k

Example:

a = (2,5,1,3,6,8,2,10,4)

Strictly increasing substrings of length 5 (maximum length):

(2,5,6,8,10)

(2,3,6,8,10)

(1,3,6,8,10)

Longest strictly increasing substring

1. Analysis of the solution structure.

   Let $s=(a_{j1}, a_{j2},\ldots,a_{j(k-1)},a_{jk})$ be an optimal solution. It means that there is no element in a[1..n] after a_{jk} (whose index is greater than jk) that is greater than a_{jk}. In addition, there is no element in the initial string with index between $j_{(k-1)}$ and j_k and value between the values of these elements of the substring s (because s would no longer be an optimal solution).

   We show that $s'=(a_{j1}, a_{j2},\ldots,a_{j(k-1)})$ is an optimal solution to the problem of determining the longest substring ending in $a_{j(k-1)}$.

   We assume that s' is not optimal. It follows that there is a substring s" of larger length. Adding to s" the element a_{jk} would obtain a better solution than s, implying that s is not optimal. This leads to a contradiction, so s' is an optimal solution to the subproblem of determining the longest increasing substring that ends in $a_{j(k-1)}$

So the problem has the optimal substructure property

Longest strictly increasing substring

2. Constructing a recurrence relation

Let Bi be the number of elements of the longest strictly increasing subsequence that ends in a_i

$$B_i = \begin{cases} 1 & i = 1 \\ 1 + \max\{B_j | i \leq j \leq i-1, a_j < a_i\} & otherwise \end{cases}$$

Example:

a = (2,5,1,11,3,6,8,2,10,4)

B = (1,2,1,3,2,3,4,2,5,3)

The longest strictly increasing substring has length 5 and ends in the element with value 10. The string is built starting from the last element. An example of such a string is (from last elements to first: 10, 8, 6, 3, 1)

Longest strictly increasing substring

3. Development of the recurrence relation

$$B_i = \begin{cases} 1 & i = 1 \\ 1 + \max\{B_j | i \leq j \leq i-1, a_j < a_i\} & otherwise \end{cases}$$

Complexity order: $\Theta(n^2)$

```
computeB(a[1..n])
  B[1]←1
  FOR i ← 2,n DO
    max ← 0
    FOR j ← 1,i-1 DO
      IF a[j]<a[i] AND max<B[j]
      THEN max ← B[j]
      ENDIF
    ENDFOR
    B[i] ← max+1
  ENDFOR
RETURN B[1..n]
```

Longest strictly increasing substring

4. Constructing the solution

- Determine the maximum of B
- Construct s successively starting from the last element

Complexity order: Θ(n)

```
Build_solution(a[1..n],B[1..n])
 m ← 1
 FOR i ← 2,n DO
  IF B[i]>B[m] THEN m ← i ENDIF
 ENDFOR
 k ← B[m]
 s[k] ← a[m]
 WHILE B[m]>1 DO
   i ← m-1
   WHILE a[i]>=a[m] OR B[i] != B[m]-1 DO
       i ← i-1
   ENDWHILE
   m ← i; k ← k-1; s[k] ← a[m]
 ENDWHILE
RETURN s[1..k]
```

Longest common substring

Given two strings (sequences) $a_1,\ldots, a_n$ and $b_1,\ldots, b_m$, determine a substring $c_1,\ldots c_k$ that satisfies:

- It is a common substring of the strings a and b, that is, there exist $i_1,\ldots,i_k$ and $j_1,\ldots,j_k$ such that $c_1=a_{i1}=b_{j1}$, $c_2=a_{i2}=b_{j2}$, … , $c_k=a_{ik}=b_{jk}$
- k is maximum (longest common substring)

Remark: this problem is a particular case encountered in bioinformatics aiming to analyze the similarity between two nucleotide strings (DNA) or amino acids – the longer they have a common substring, the more similar the two initial strings are

Longest common substring

Example

a: 2 1 4 3 2

b: 1 3 4 2

Common substrings

1, 3

1, 2

4, 2

1, 3, 2

1, 4, 2

Variant of the problem: determining the longest common subsequence of consecutive elements

Example

a: 2 1 3 4 5

b: 1 3 4 2

Common substrings

1, 3

3, 4

1, 3, 4

Longest common substring

1. Analysis of the structure of an optimal solution

   Let P(i,j) be the problem of determining the longest common substring of the strings a[1..i] and b[1..j]. If a[i]=b[j] then the optimal solution contains this common element and the rest of the elements are represented by the optimal solution of the subproblem P(i-1,j-1) (which consists of determining the longest common substring of the strings a[1..i-1] and b[1..j-1, respectively). If a[i] is different from b[j] then the optimal solution coincides with the best of the solutions of the subproblems P(i-1,j) and P(i,j-1, respectively).

2. Derivation of the recurrence relation. Let L(i,j) be the length of the optimal solution to the problem P(i,j). Then:

$$L[i] = \begin{cases} 0 & j = 0 \; or \; j = 0 \\ 1 + L[i-1, j-1] & a[i] = b[j] \\ \max\{L[i-1, j], L[i, j-1]\} & otherwise \end{cases}$$

Longest common substring

Exemple:

a: 2 1 4 3 2

b: 1 3 4 2

$$L[i] = \begin{cases} 0 & j = 0 \; or \; j = 0 \\ 1 + L[i-1, j-1] & a[i] = b[j] \\ \max\{L[i-1, j], L[i, j-1]\} & otherwise \end{cases}$$

j→		-	1	3	4	2
i↓		0	1	2	3	4
0	-	0	0	0	0	0
1	2	0	0	0	0	1
2	1	0	1	1	1	1
3	4	0	1	1	2	2
4	3	0	1	2	2	2
5	2	0	1	2	2	3

Longest common substring

Exemple:

a: 2 1 4 3 2

b: 1 3 4 2

$$L[i] = \begin{cases} 0 & j = 0 \; or \; j = 0 \\ 1 + L[i-1, j-1] & a[i] = b[j] \\ \max\{L[i-1, j], L[i, j-1]\} & otherwise \end{cases}$$

```
build(a[1..n],b[1..m])
 FOR i ← 0,n DO L[i,0] ← 0 ENDFOR
 FOR j ← 1,m DO L[0,j] ← 0 ENDFOR
 FOR i ← 1,n DO
  FOR j ← 1,m DO
    IF a[i]==b[j]
      THEN L[i,j] ← L[i-1,j-1]+1
    ELSE
      L[i,j] ← max(L[i-1,j],L[i,j-1])
    ENDIF
  ENDFOR
ENDFOR
RETURN L[0..n,0..m]
```

Longest common substring

Building the solution (recursive version):

```
Build_solution(i,j)
IF L[i,j]>0 THEN
  IF a[i]==b[j] THEN
    Build_solution(i-1,j-1)
    k ← k+1
   c[k] ← a[i]
  ELSE
    IF L[i-1,j]>L[i,j-1]
    THEN Build_solution(i-1,j)
    ELSE Build_solution (i,j-1)
ENDIF ENDIF ENDIF
```

Ideea:

At position (i, j):

If a[i-1] == b[j-1]:

the element is part of the LCS

add it and go diagonally: (i-1, j-1)

If they are different:

go to the cell that has the higher value of:

L[i-1][j] (top)

L[i][j-1] (left)

Remarks:

- a, b, c and k are global variables
- Before calling the function, the variable k is initialized (k=0)
- The build function is called by build(n,m)

Memoization technique

The table is initialized with a virtual value (this value must be different from any value that would be obtained by expanding the recurrence relation)

The target value (e.g. V[n,C]) is calculated recursively, but all intermediate values are stored and used when necessary.

Remark: v[1..n], d[1..n] and V[0..n,0..C] are global variables.

Call: comp(n,C)

- Initialisation of virtual vales

```
FOR i←0,n DO
    FOR j←0,C DO V[i,j] ←-1 ENDFOR
ENDFOR
```

- Recursive call

```
comp(i,j)
  IF i=0 OR j=0 THEN V[i,j] ←0; RETURN V[i,j]
  ELSE
     IF V[i,j] != -1 THEN RETURN V[i,j]
     ELSE
          IF j<d[i] THEN V[i,j] ←comp(i-1,j)
           ELSE V[i,j] ← max(comp(i-1,j),comp(i-1,j-d[i])+v[i])
       ENDIF
       RETURN V[i,j]
    ENDIF
 ENDIF
```

Memoization technique

Purpose: only subproblems whose solution is included in the solution of the initial problem are solved (in addition, a subproblem is solved only once)

Idea: the top-down approach is combined with the bottom-up approach

Motivation:

- The classic top-down approach solves only subproblems that contribute to the solution of the problem, but a subproblem is solved each time it appears (for this reason, recursive implementation is generally inefficient)
- The classic bottom-up approach solves all subproblems (even those that do not contribute to the optimal solution) but each problem is solved only once
- Memoization technique solves only subproblems that contribute to the solution of the problem

Memoization and Fibonacci

- Before recursive code below called, must initialize results[] so all values are -1

```
fib_mem(n, results[])
    IF ( results[n] != -1 THEN
        return results[n];  // return stored value

    if ( n == 0 || n ==1 ) val = n; // odd but right
    else
        val = fib_mem(n-1, results) + fib_mem(n-2, results);
    results[n] = val; // store calculated value
    return val;
```

Quiz

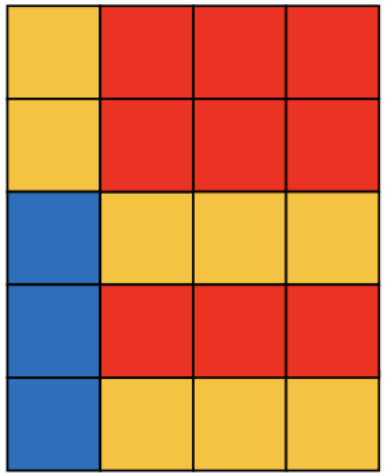
Back to the FloodIt Game (filling the grid with the same color in as few steps as possible)

What color would you start with?

a) Red

b) Yellow

c) Blue



Q&A