Bioinformatics Algorithms

Dynamic Programming

RNDr. David Hoksza, Ph.D.
http://siret.cz/hoksza

Outline

- Dynamic programming basics
  - recursion
  - approaches
- Example problems
  - Fibonacci numbers
  - matrix product
  - longest common subsequence

Sources

- Animations
  - AlViE suite - https://sites.google.com/site/alviehomepage/

Recursion

- Method where the **solution** to a problem **is built** from solution of **smaller instances** of the same problem

- Usually recursive problem is solved by a parameterized function which calls itself with parameter reflecting smaller instance(recursion)

- Examples
    - factorial
        - $\boldsymbol{n}! = \boldsymbol{n}(\boldsymbol{n}-\mathbf{1})! \rightarrow \boldsymbol{F}(\boldsymbol{n}) = \boldsymbol{n}\boldsymbol{F}(\boldsymbol{n}-\mathbf{1})$
    - Fibonacci numbers
    - greatest common divisor
    - binary search
    - ...

Dynamic Programming (DP)

- Algorithm **design technique**/concept

- Conditions
  - the optimal solutions to a problem is **composed of optimal solutions to subproblems**
  - if there are several optimal solutions, **we don't care which one we get**

- Approaches
  - Top-down
    - retains standard recursive top-down structure but stores
  - **Bottom-up**
    - higher levels share results from the lower levels
    - DP solutions are often considered only those using bottom-up approach

  - *see Fibonacci for an example*

Dynamic Programming and Dimension

- Dynamic programming solves problems by dividing problem into subproblems and using their results later

- **Memoization**
  - to store results of the subproblems n-dimensional array is usually used

- **Usually**, we talk about dynamic programming when $\boldsymbol{n \geq 2}$
  - e.g., algorithm for computation of Fibonacci numbers fulfills the conditions of DP solution but $n < 2$

Fibonacci Numbers (FN)

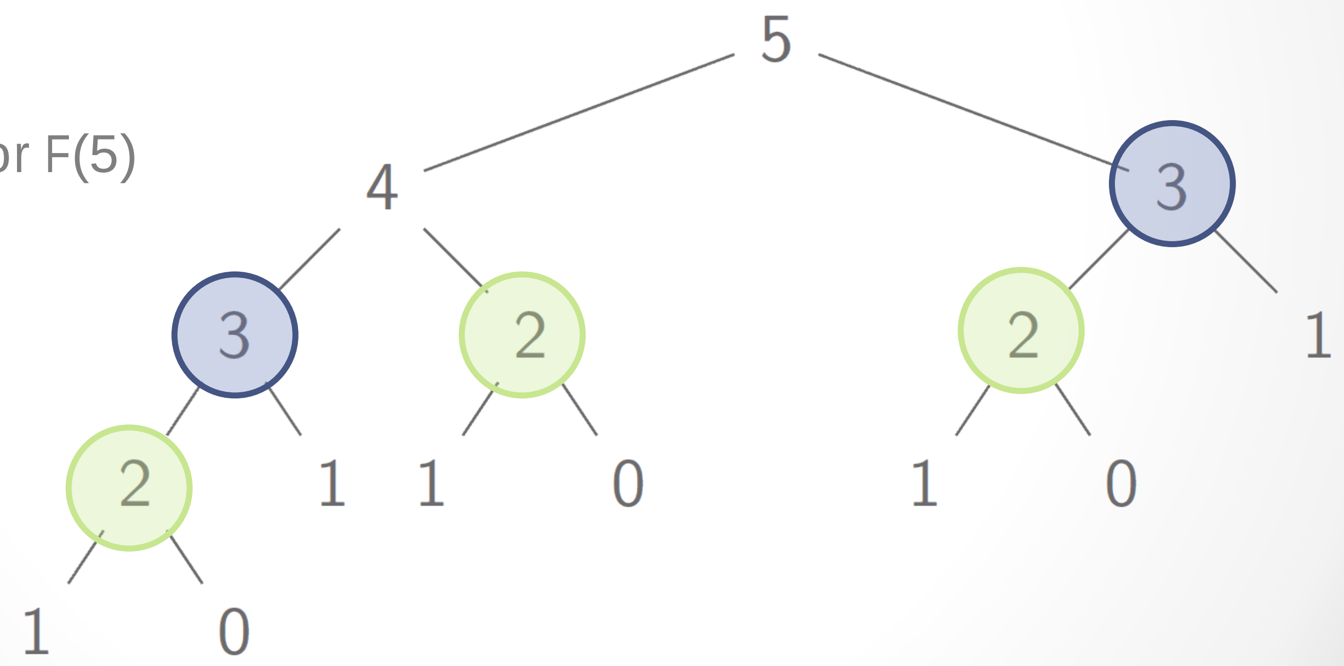
- **Definition**
  - $\boldsymbol{F(n)} = \begin{cases} \boldsymbol{n} & \boldsymbol{n \le 1} \\ \boldsymbol{F(n-1) + F(n-1)} & \boldsymbol{n > 1} \end{cases}$
  - $F(0) = 0, F(1) = 1, F(2) = 1, F(3) = 2, F(4) = 3, F(5) = 5, F(6) = 8, \ldots$
- **Task**
  - Given $\boldsymbol{n}$ compute F($\boldsymbol{n}$)

FN - Recursion

```
static int FibRec(int n)
{
    if (n <= 1) return n;
    else return FibRec(n - 1) + FibRec(n - 2);

}
```

- example for F(5)



- high redundancy

FN – DP – Top-Down

```
static int FibDPTD_rec(int n, int[] f)
{
    if (n <= 1) return n;
    else if (f[n] == 0)
    {
        f[n] = FibDPTD_rec(n - 1, f) + FibDPTD_rec(n - 2, f);
    }
    return f[n];
}

static int FibDPTD(int n)
{
    //f ... dynamic programming array
    int[] f = new int[n];
    Array.Clear(f, 0, f.Length);
    return FibDPTD_rec(n, f);
}
```

- requires O(n) time and space

FN – DP – Bottom-Up

```
static int FibDP(int n)
{
    //f ... dynamic programming array
    f[0] = 0;
    f[1] = 1;
    for (int k = 2; k < n; k++) f[k] = f[k - 1] + f[k - 2];
    return f[n];
}
```

- one-dimensional dynamic programming array
- no redundant computations
- for computation of the solution, the subsolutions are used

- requires O(n) time and space (can be done in O(1))

Matrix Product Ordering (MPO)

- **Matrix multiplication**
  - o takes a pair of matrices $A[p \times q]$ and B$[q \times r]$
  - o and produces matrix C$[p \times r]$

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix}}_{4\times 2 \text{ matrix}} \underbrace{\begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix}}_{2\times 3 \text{ matrix}} = \underbrace{\begin{bmatrix} \cdot & x_{12} & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & x_{33} \\ \cdot & \cdot & \cdot \end{bmatrix}}_{4\times 3 \text{ matrix}}$$

$$x_{12} = (a_{11}, a_{12}) \cdot (b_{12}, b_{22}) = a_{11}b_{12} + a_{12}b_{22}$$

$$x_{33} = (a_{31}, a_{32}) \cdot (b_{13}, b_{23}) = a_{31}b_{13} + a_{32}b_{23}.$$

  - o associative
    - $A(BC) = (AB)C$
    - $ABCD = ((AB)C)D = (A(BC)D) = A((BC)D) = A(B(CD)) = (AB)(CD)$

- **Task**
  - o Given $\boldsymbol{n}$ matrixes $\boldsymbol{A_1, A_2, \dots A_n}$ find such a **parenthesization minimizing number of multiplications** of the matrixes' items
    - order of multiplication in the chain is important
    - $n = 3, A_1[2 \times 3], A_2[3 \times 2], A_3[2 \times 5]$
      - o A_1A_2= 12 multiplications, $(A_1A_2)A_3$ = 20 multiplications → 32 in total
      - o A_2A_3= 30 multiplications, $A_1(A_2A_3)$ = 30 multiplications → 60 in total

MPO - Recursion

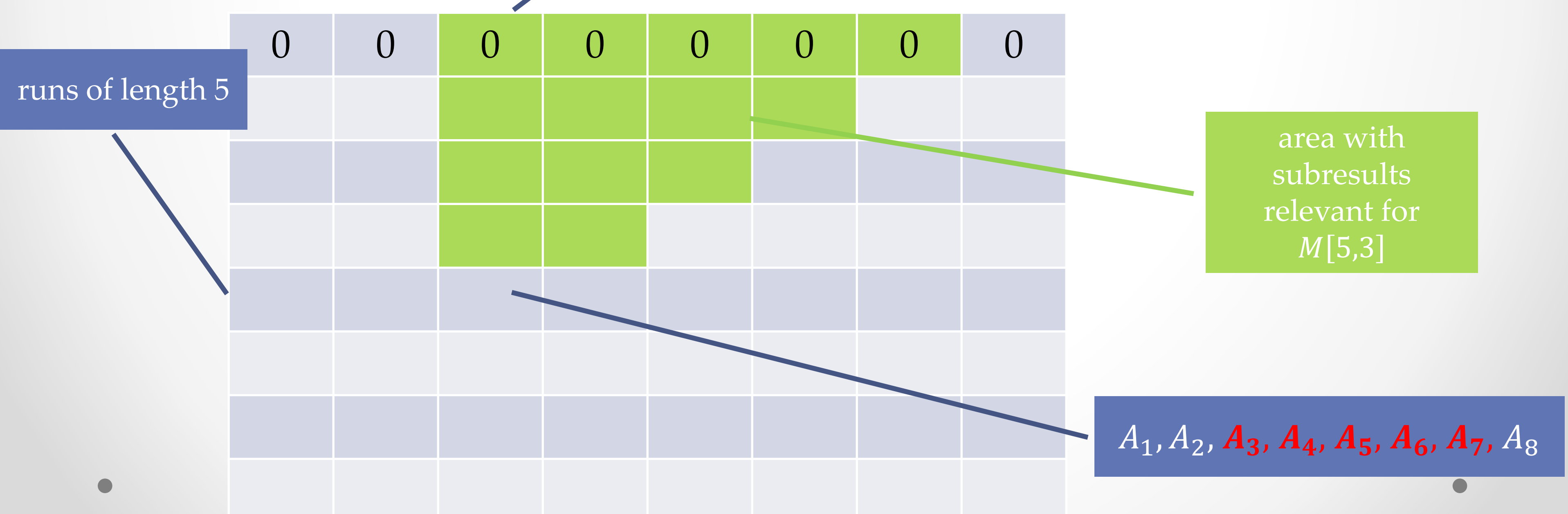
```
static int MPORec(int ixFrom, int ixTo)
{   //p ... matrices dimenstions
    int cntMin = 0;
    if (ixFrom != ixTo)
    {
        cntMin = Int32.MaxValue;
        for (int k = ixFrom; k < ixTo; k++)
        {
            int cntMult = MPORec(ixFrom, k) + MPORec(k+1, ixTo) +
                p[ixFrom - 1] * p[k] * p[ixTo];
            if (cntMult < cntMin) cntMin = cntMult;
        }
    }
    return cntMin;
}
```

- MPORec returns minimum number of multiplications needed to multiply matrices $A_{ixFrom+1}, A_{ixTo+1}$(+1 because C# arrays go from 0)
- scan the array of matrices
- at each position take the best result from left, right and add their product cost
- parenthesization can be stored in an auxiliary structure

MPO - DP

- DP matrix M will store in $\boldsymbol{M[i,j]}$ minimum number of multiplications needed to multiply i consequent matrices starting at position j $(\boldsymbol{A_j, A_{j+1}, \dots A_{j+i-1}}) \rightarrow \boldsymbol{M[n,1]}$ contains the **result** (indexing from 1 for sake of clarity)

- $M[i,j]$ can be computed by increasing i
  - $M[1,j] = 0$
  - $M[i,j] = \min(M[k,j] + M[i-k, j+k] + p[j-1] \times p[j+k-1] \times p[j+i-1])$



MPO – DP (cont.)

```
static int MPO()
{
    //n ... # matrices
    //m ... dynamic programming matrix
    //m[i][j] = minimum number of operations needed to multiply A_i, ..., A_i+j
    //p ... matrices dimenstions
    for (int i = 1; i <= n; i++) m[1][i] = 0; //indexing from 1 for sake of clarity
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n - i +1; j++)
        {
            cntMin = Int32.MaxValue;
            for (int k = 1; k < i - 1; k++)
            {
                int cntMult = m[k][j] + m[i - k][j + k] + p[j - 1] * p[j + k - 1] * p[j + i - 1];
                if (cntMult < cntMin) cntMin = cntMult;
            }
            m[i][j] = cntMin;
        }
    }
    return cntMin;
}
```

- Runs in $O(n^3)$ time and $O(n^2)$ space

Longest Common Subsequences (LCS)

- Task
  - o Given **two sequence** $S_1[1 \ldots m]$ and $S_2[1 \ldots n]$, find a **longest common subsequence** (not substring) common to both.
    - Sequence $\boldsymbol{S}$ of length $|S|$ is a **subsequence** of sequence **A** of length $|A|$ if there exists indeces $1 \le i_1 < i_2 < \ \ldots \ < i_{|S|} <= |A|$ in A such that $\boldsymbol{S}[\boldsymbol{j}] = \boldsymbol{A}[\boldsymbol{i_j}]$, $j \ = \ 0, 1, \ldots, |S|$
    - $\boldsymbol{S}$ is a **common subsequence** of S_1 and S_2 only if it is subsequence of both S_1 and S_2 .
- Example
  - A B C B D A B
  - B D C A B A
- Particularly important in computational biology
-

LCS – Brute-Force

- Check every subsequence of $S_{1\ \mathrm{in}}\ S_2$
  - for each subsequence ss_1 in s_1 we can test whether it is present in s_2 in O(n) time
    - scanning s_2 linearly and checking whether first letter in s_2 corresponds to the first letter in ss_1
    - if so, let us continue in the same fashion with the second letter from that position
    - when we run out of letters of ss_1, ss_1 is present in s_2
  - the are $O(2^m)$ subsequences → complexity of the brute-force algorithm is $O(n2^m)$
    - OK for short sequences but not for, e.g., DNA sequences

LCS – DP

- DP solution is based on computing LCS for prefixes of S_1 and S_2 (subproblems in DP)

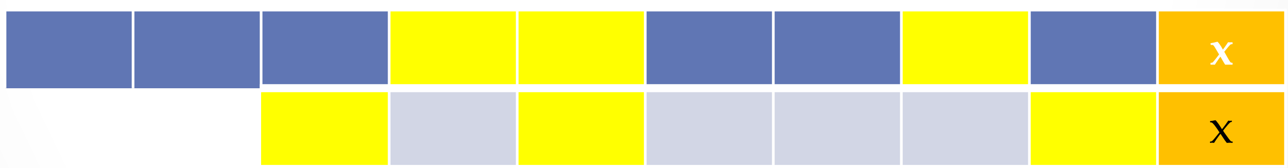
- Let us denote $LCS(i, j)$ LCS of i and j long prefixes of S_1 and S_2
  - $LCS(|S_1|, |S_2|)$ = solution of LCS

- Recursive rule
  - $LCS(i, j) \begin{cases} 0 & i = 0 \; or \; j = 0 \\ L(i, -1 \, j - 1) + 1 & S_1[i] = S_2[j] \\ \max\{L(i, j - 1), L(i - 1, j)\} & S_1[i] \neq S_2[j] \end{cases}$

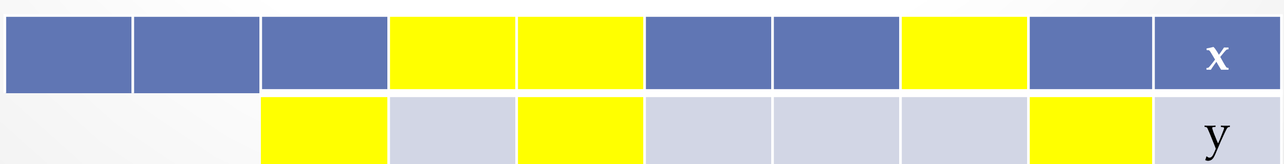
LCS – DP (cont.)

- $$L(i,j)\begin{cases}0 & i = 0 \; or \; j = 0 \quad \mathbf{(1)} \\ L(i,-1\,j-1)+1 & S_1[i] = S_2[j] \quad \mathbf{(2)} \\ \max\{L(i,j-1), L(i-1,j)\} & S_1[i] \neq S_2[j] \quad \mathbf{(3)}\end{cases}$$

1. If there is only one sequence, LCS = 0
2. If $\mathrm{LCS}(A_1, A_2) = n$ then $\mathrm{LCS}(A_1 x, A_2 x) = n + 1$
   - because if two sequences have the same 1-letter suffix then their LCS will contain it, otherwise it wouldn't be LCS



3. If two sequences $A_1 x$ and $A_2 y$ differ at last position then their LCS is identical to either $\mathrm{LCS}(A_1 x, A_2)$ or $\mathrm{LCS}(A_1, A_2 y)$



LCS – DP (cont.)

```
LCS( a, b )
  FOR (i = 0; i <= m; i = i+1)
    length[i][0] = 0;
  FOR (j = 0; j <= n; j = j +1)
    length[0][j] = 0;
  FOR (i = 1; i <= m; i = i+1)
    FOR (j = 1; j <= n; j = j+1) {
      IF (a[i-1] == b[j-1]) {
        length[i][j] = length[i-1][j-1] + 1;
      } ELSE IF (length[i][j-1] > length[i-1][j]) {
        length[i][j] = length[i][j-1];
      } ELSE {
        length[i][j] = length[i-1][j];
      }
    }
  RETURN length[m][n];
```

- The alignment itself can be identified easily by backtracking

LCS Matrix

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

LCS Backtracking

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

	B	D	C			A	B	A
A	B		C	B	D	A	B	

			B	D	C	A	B	A
A	B	C	B	D		A	B	

LCS - example

- AlVie

- Practise
  - LCS (HUMAN, CHIMPANZEE)