ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Algorithm Theory

11 Dynamic Programming

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik

Wintersemester 2007/08

CoNe Freiburg

IIF INSTITUT FÜR INFORMATIK FREIBURG

Outline

- **General approach, differences to a recursive approach**
- **Basic example: Computation of the Fibonacci numbers**

Method of Dynamic Programming

- **Recursive approach**
  - Solve a problem by solving several smaller analogous subproblems of the same type.
  - Then combine these solutions to generate a solution to the original problem.
- **Drawback: Repeated computation of solutions**
- **Dynamic programming**
  - Once a subproblem has been solved, store its solution in a table so that it can be retrieved later by simple table lookup

Example: Fibonacci Numbers

f(0) = 0
f(1) = 1
f(*n*) = *f*(*n* – 1) + *f*(*n* – 2), falls *n* ≥ 2

Remark:

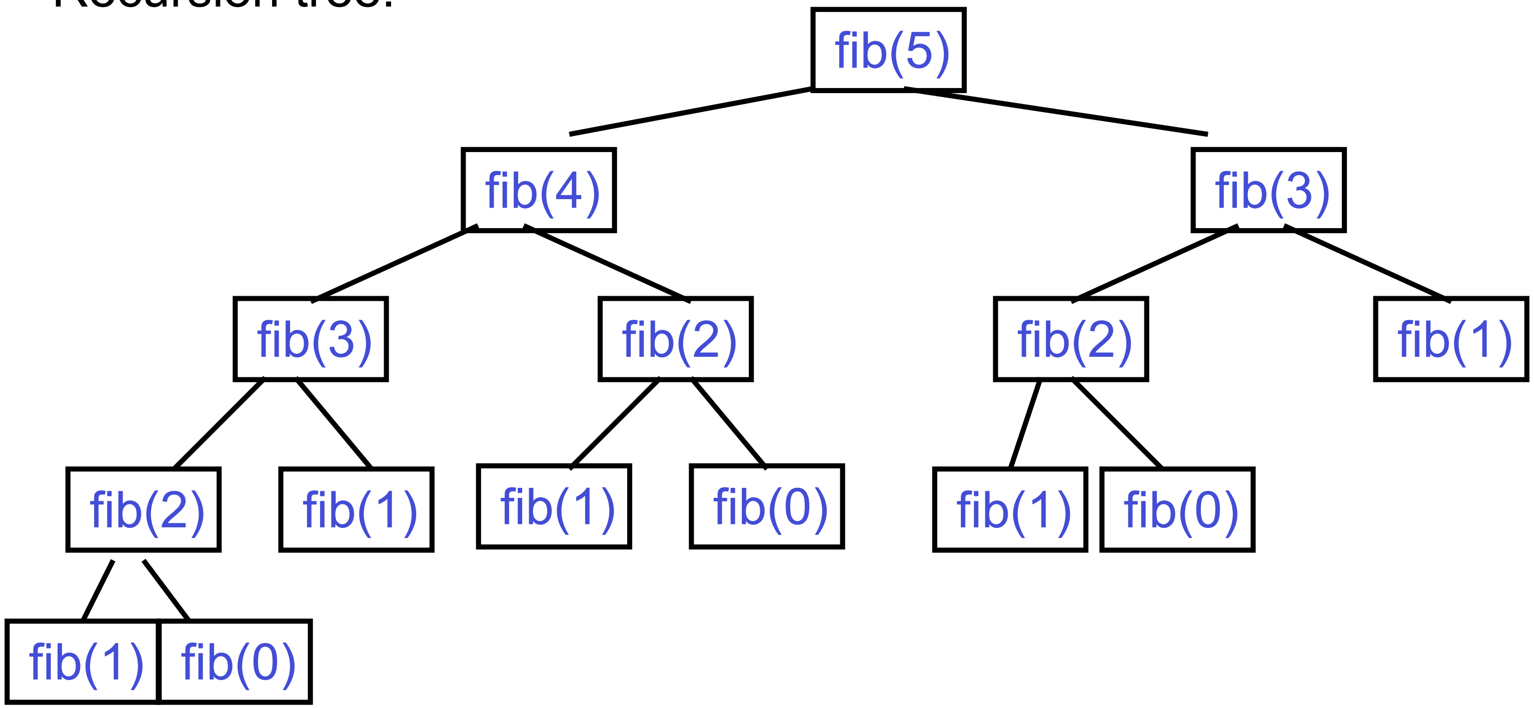
$$f(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \qquad \phi = \frac{1+\sqrt{5}}{2}$$

Straightforward Implementation:

```
procedure fib (n : integer) : integer
if (n == 0) or (n ==1)
        then return n
        else return fib(n – 1) + fib(n – 2)
```

Example: Fibonacci Numbers

Recursion tree:



Repeated computation!

$$T(n) \geq f(n) \geq 2^n$$

Dynamic Programming

- **Approach:**
  1. Recursively define problem P.
  2. Determine a set T consisting of all subproblems that have to be solved during the computation of a solution of P.
  3. Find an order T_0 , ..., T_k of the subproblems in T such that during the computation of a solution to T_i only subproblems T_j with $j < i$ arise.
  4. Solve T_0 ,...,T_k in this order and store the solutions.

Example: Fibonacci Numbers

1. **Recursive definition of the Fibonacci numbers, based on the standard definition**
2. $T = \{ f(0),..., f(n\text{-}1)\}$
3. $T_i = f(i), \quad i = 0,...,n - 1$
4. **Computation of** *fib*(*i*)**, for** ***i≥2,*** **only requires the results of the last two subproblems** *fib(i-1)* **and** *fib(i-2).*

Example: Fibonacci Numbers

Computation by dynamic programming, version 1

procedure *fib*(*n* : *integer*) : *integer*

f_0 **:= 0;** f_1 **:= 1**

for k **:= 2 to** n **do**

  $f_k := f_{k-1} + f_{k-2}$

return f_n

Example: Fibonacci Numbers

Computation by dynamic programming, version 2

procedure *fib* (*n* : *integer*) : *integer*

$f_{next\text{-}to\text{-}last}$ **:= 0;** f_{last} **:=1**

for k **:= 2 to** n **do**

    $f_{current} := f_{last} + f_{next\text{-}to\text{-}last}$

    $f_{next\text{-}to\text{-}last} := f_{last}$

    $f_{last} := f_{current}$

if $n \leq 1$ **then return** n **else return** $f_{current}$ **;**

Linear running time, constant space requirement!

Computation of the Fibonacci Numbers using Memoization

Compute each number exactly once, store it in an array *F*[0...*n*]:

```
procedure fib (n : integer) : integer
F[0] := 0;   F[1] := 1;
for i :=2 to n do
      F[i] := ∞;
return lookupfib(n)
```

The procedure *lookupfib* is defined as follows:

```
procedure lookupfib(k : integer) : integer
if F[k] < ∞
    then return F[k]
    else F[k] := lookupfib(k – 1) + lookupfib(k – 2);
            return F[k]
```

Optimal Substructure

Dynamic programming is typically applied to

optimization problems.

An optimal solution to the original problem contains

optimal solutions to smaller subproblems.

Matrix Chain Multiplications

Given: sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ **of matrices**

Goal: compute the product $A_1 \cdot A_2 \cdot \ldots . \cdot A_n$

Problem: Parenthesize the product in a way that minimizes the number of scalar multiplications.

Definition: A product of matrices is ***fully parenthesized*****, if it is either a single matrix or the product of two fully parenthesized matrix surrounded by parentheses.**

Examples of Fully Parenthesized Matrix Products

All possible fully parenthesized matrix products of the chain $\langle A_1, A_2, A_3, A_4 \rangle$ are:

$$(A_1(A_2(A_3A_4)))$$

$$(A_1((A_2A_3)A_4))$$

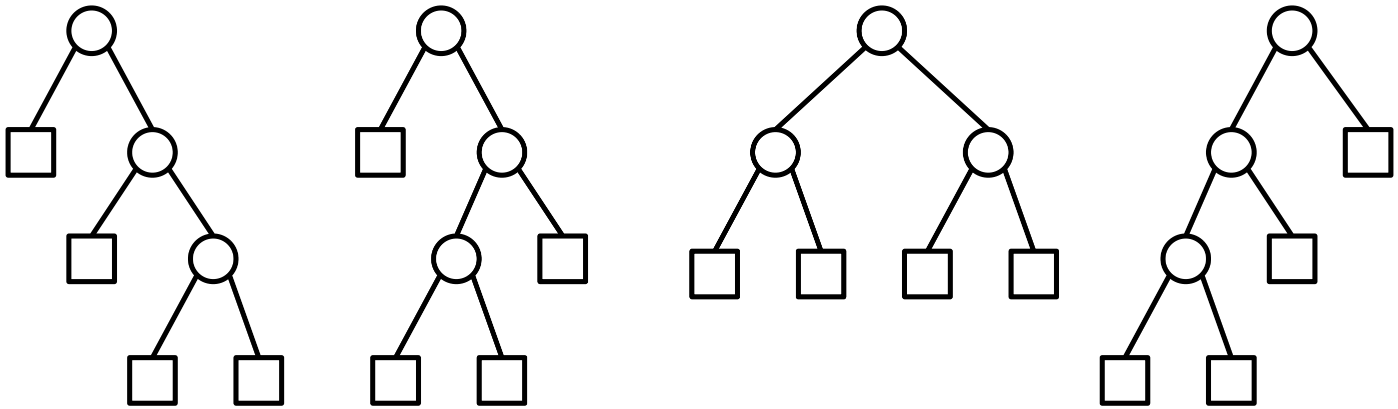
$$((A_1A_2)(A_3A_4))$$

$$((A_1(A_2A_3))A_4)$$

$$(((A_1A_2)A_3)A_4)$$

Number of Different Parenthesizations

Different parenthesizations corresponds to different trees:



Number of Different Parenthesizations

P(n) be the number of alternative parenthesizations of the product $A_1 \ldots A_k A_{k+1} \ldots A_n$

$$P(1) = 1$$

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \quad \text{for} \quad n \geq 2$$

$$P(n+1) = \frac{1}{n+1}\binom{2n}{n} = \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4^n}{\sqrt{n^5}}\right)$$

$$P(n+1) = C_n \quad n\text{-th Catalan number}$$

Determining the optimal parenthesization by exhaustive search is not reasonable.

Multiplication of two Matrices

$$A = (a_{ij})_{p \times q},\ B = (b_{ij})_{q \times r},\ A \times B = C = (c_{ij})_{p \times r}.$$

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

Algorithm *Matrix-Mult*
Input: ($p \times q$) matrix A, ($q \times r$) matrix B
Output: ($p \times$ r) matrix C = A · B

```
for i := 1 to p do
    for j :=1 to r do
        C[i, j] := 0
        for k := 1 to q do
            C[i, j] := C[i, j] + A[i, k] ·B[k,j]
```

Number of multiplications and additions: $p \cdot q \cdot r$

Using this algorithm, multiplying two $(n \times n)$ matrices requires n^3 multiplications.
Remark: This can be also done using $O(n^{2.376})$ multiplications.

Matrix Chain Multiplication: Example

- **Computation of the product $A_1A_2A_3$, where**
- **A_1 : 10 × 100 matrix**
- **A_2 : 100 × 5 matrix**
- **A_3 : 5 × 50 matrix**
- **Parenthesization $(A_1 A_2) A_3$ requires**
  - $A' = (A_1 A_2)$:
  - $A' A_3$:
  - Sum:

Matrix Chain Multiplication: Example

- **Computation of the product $A_1A_2A_3$, where**
- **A_1 : 10 × 100 matrix**
- **A_2 : 100 × 5 matrix**
- **A_3 : 5 × 50 matrix**
- **Parenthesization (A_1 (A_2 A_3)) requires**
  - A''= (A_2 A_3):
  - A_1 A'' :
  - Sum:

Structure of an Optimal Parenthesization

- **($A_{i...j}$) = (($A_{i...k}$) ($A_{k+1....j}$)) $i \leq k < j$**
  - Any optimal solution to the matrix-chain multiplication problem solutions to subproblems.
- **Determining an optimal recursively**
  - Let m[i,j] be the minimum number of operations needed to compute the product $A_{i...j}$:
  - m[i,j] = 0 if i = j
  - m[i,j] = $m[i,j] = \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}$
  - s[i,j] = optimal splitting value k
    - the optimal parenthesization of ($A_{i...j}$) splits the product between A_k and A_{k+1}

Recursive Matrix Chain Multiplication

Algorithm *rec-mat-chain*(*p, i, j*)

Input: sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p_{i-1} \times p_i$ is the dimensions of matrix A_i

Invariant: rec-mat-chain(*p, i, j*) returns $m[i, j]$

1 if $i = j$ then return 0

2 m[i, j] := ∞

3 for $k := i$ to $j - 1$ do

4 $m[i, j] := \min(m[i,j],\ p_{i-1}\, p_k\, p_j\ +$
rec-mat-chain(p, i, k) +
rec-mat-chain($p, k+1, j$))

5 return $m[i, j]$

Initial call: rec-mat-chain(p, 1, n)

Recursive Matrix Chain Multiplication — Runtime

Let $T(n)$ be the time taken by rec-mat-chain(p,1,n).

$$T(1) \geq 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + 1\right)$$

$$\geq n + 2\sum_{i=1}^{n-1} T(i)$$

$$\Rightarrow T(n) \geq 3^{n-1} \qquad \text{(induction)}$$

Exponential runtime!

Matrix Chain Multiplication — Dynamic Programming

Algorithmus *dyn-mat-chain*

Input: sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ $\quad p_{i-1} \times p_i$ dimension of matrix A_i

Output: $m[1,n]$

```
n := length(p)
for i := 1 to n do m[i, i] := 0
for l := 2 to n do                          /* l = length of the subproblem */
  for i := 1 to n – l + 1 do                /* i is the left index */
      j := i + l – 1                        /* j is is the right index*/
      m[i, j] := ∞
      for k := i to j - 1 do
          m[i, j] := min(m[i, j], p_{i-1} p_k p_j + m[i, k] + m[k + 1, j])
return m[1, n]
```

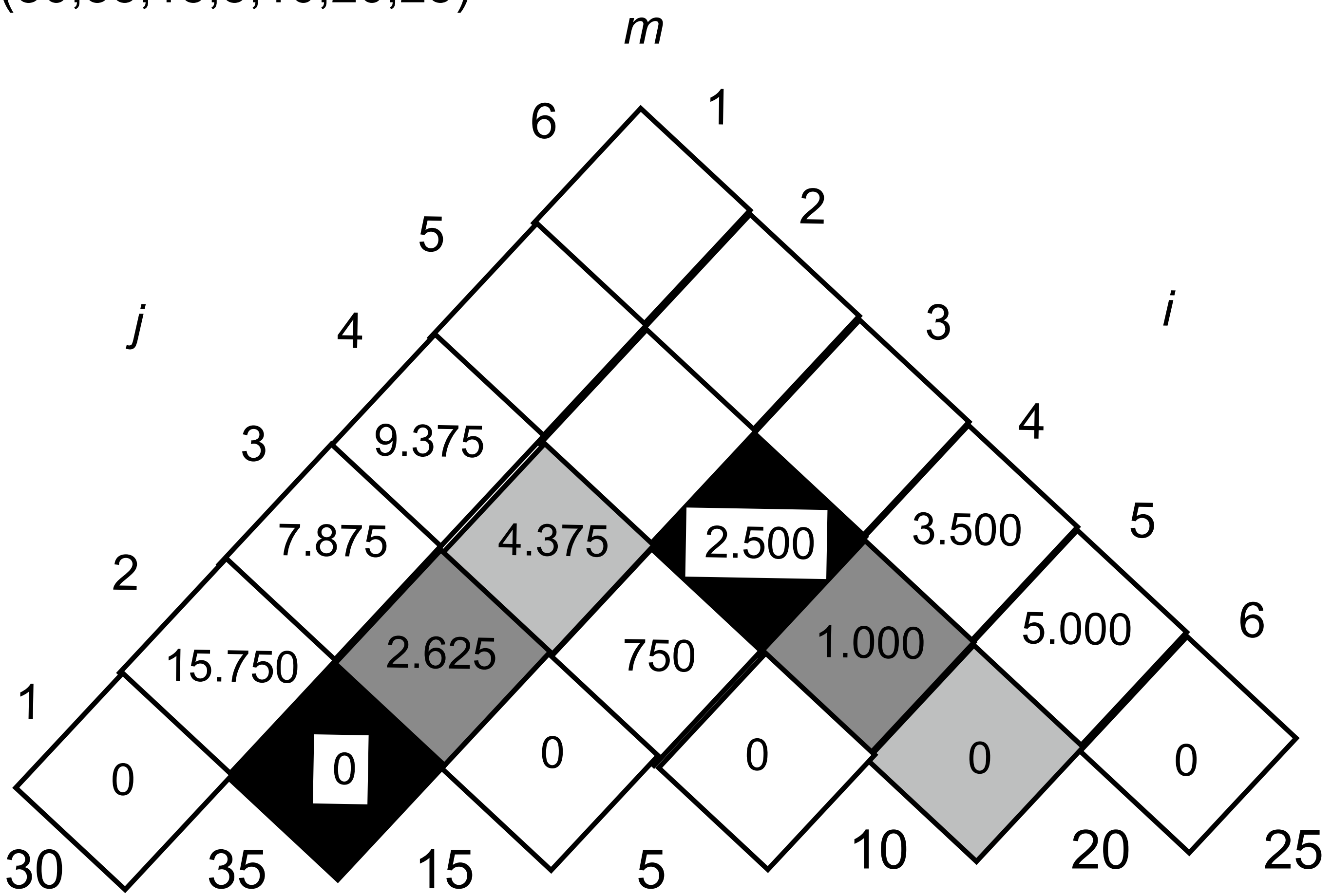
Example

A_1	30×35	A_4	5×10
A_2	35×15	A_5	10×20
A_3	15×5	A_6	20×25

P = (30,35,15,5,10,20,25)

Example

P = (30,35,15,5,10,20,25)



Example

$$
\begin{aligned}
m[2,5] &= \min_{2\le k<5}\{m[2,k]+m[k+1,5]+p_1p_kp_5\} \\
&= \min\left\{\begin{array}{c} m[2,2]+m[3,5]+p_1p_2p_5 \\ m[2,3]+m[4,5]+p_1p_3p_5 \\ m[2,4]+m[5,5]+p_1p_4p_5 \end{array}\right\} \\
&= \min\left\{\begin{array}{c} 0+2,500+35\cdot 15\cdot 20 \\ 2,625+1,000+35\cdot 5\cdot 20 \\ 4,375+0+35\cdot 10\cdot 20 \end{array}\right\} \\
&= \min\left\{\begin{array}{c} 13,000 \\ 7,125 \\ 11,375 \end{array}\right\} \\
&= 7,125
\end{aligned}
$$

Matrix Chain Multiplication and Optimal Splitting Values using Dynamic Programming

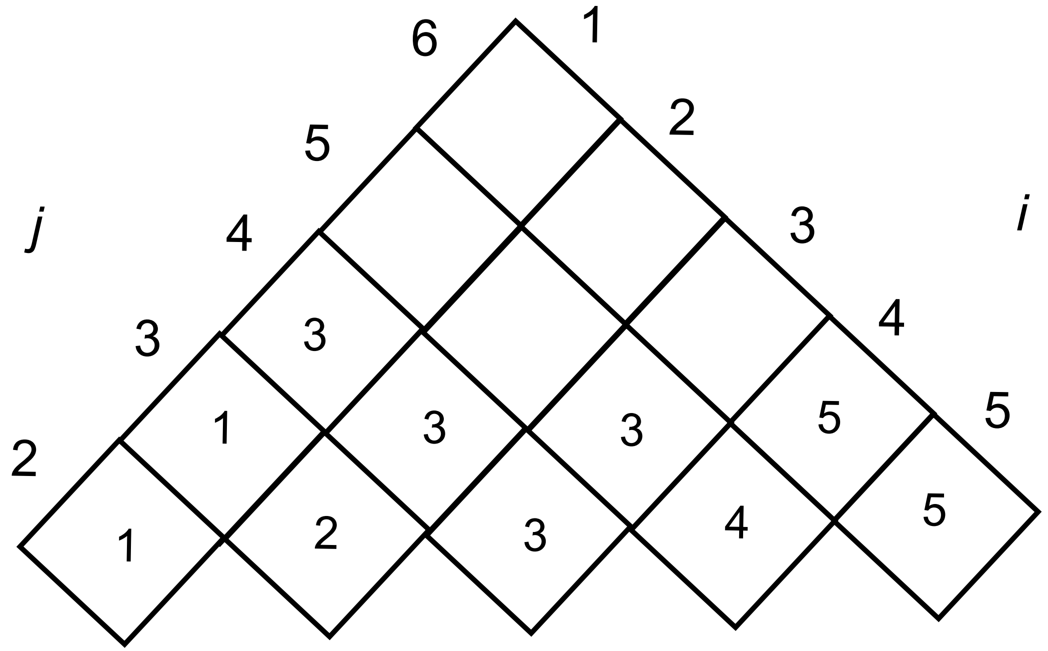
Algorithm *dyn-mat-chain(p)*

Input: sequence $p = \langle p_0, p_1, ..., p_n \rangle$ $p_{i-1} \times p_i$ the dim. of matrix A_i

Output: $m[1,n]$ and a matrix $s[i,j]$ containing the optimal splitting values

```
n := length(p)
for i := 1 to n do m[i, i] := 0
for l := 2 to n do
    for i := 1 to n – l +1 do
        j := i + l – 1
        m[i, j] := ∞
        for k := i to j – 1 do
            q := m[i, j]
            m[i, j] := min(m[i, j], p_{i-1} p_k p_j + m[i, k] + m[k + 1, j])
            if m[i, j] < q then s[i, j] := k
return (m[1, n], s)
```

Example of Splitting Values



Computation of an Optimal Parenthesization

Algorithm *Opt-Parenths*

Input: chain *A* of matrices, matrix s containing the optimal splitting values, two indices *i* and *j*

Output: ***an optimal parenthesization of*** $A_{i...j}$

```
if i < j
   then X := Opt-Parenths(A, s, i, s[i, j])
        Y := Opt-Parenths(A, s, s[i, j] + 1, j)
        return (X·Y)
   else return A_i
```

Inital call: *Opt–Parenths*(*A*, *s*, **1**, *n*)

Matrix Chain Multiplications using Dynamic Programming — Top Down

„Memoization“ for increasing the efficiency of a recursize solution**:**

Only the *first time*, a subproblem is encountered, its solution is computed and then stored in a table

Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned (without repeated computation!)

Memoized Matrix Chain Multiplication

m[i,j] initialized with ∞

Algorithm *mem-mat-chain***(***p, i, j***)**

Invariant: *mem-mat-chain*(*p, i, j*) returns *m*[*i, j*];
    the value is correct if $m[i, j] < \infty$

```
if i = j then return 0
if m[i, j] < ∞ then return m[i, j]
for k := i to j – 1 do
      m[i, j] := min(m[i, j], p_{i-1} p_k p_j +
                     mem-mat-chain(p, i, k) +
                     mem-mat-chain(p, k + 1, j))
return m[i, j]
```

Memoized Matrix Chain Multiplication

Call:

```
n:= length(p) – 1
for i := 1 to n do
    for j := 1 to n do
        m[i, j] := ∞
mem-mat-chain(p,1,n)
```

The computation of all entries $m[i, j]$ using mem-mat-chain takes **O(n^3)** time.

O(n^2) entries

each entry $m[i, j]$ is only computed once

each entry $m[i, j]$ is looked up during the computation of $m[i',j']$ if $i' = i$ and $j' > j$ or $j' = j$ and $i' < i$

→ $m[i, j]$ is looked up for at most $2n$ entries

Final Remarks about Matrix Chain Multiplication

1. **There is an algorithm that determines an optimal parenthesization in time O(*n* log *n*)**
2. **There is a linear time algorithm that determines a parenthesization using at most 1.155 M_{opt} multiplications.**

Method of Dynamic Programming

- **Recursive approach**
  - Solve a problem by solving several smaller analogous subproblems of the same type.
  - Then combine these solutions to generate a solution to the original problem.
- **Drawback: Repeated computation of solutions**
- **Dynamic programming**
  - Once a subproblem has been solved, store its solution in a table so that it can be retrieved later by simple table lookup

Two Different Approaches

- **Bottom-up:**
  - \+ the table is maintained in an efficient way, time saving
  - \+ subproblems are solved in a special, optimized order, space saving
  - \- extensive rewriting of the original problem code is necessary
  - \- possibly, unnecessary subproblems are solved
- **Top-down (memoization)**
  - \+ only slight modifications in the original program code are necessary
  - \+ only those subproblems definitely required are solved
  - \- separate table management is time consuming
  - \- table size is often suboptimal

Optimal Substructure

Dynamic programming is typically applied to

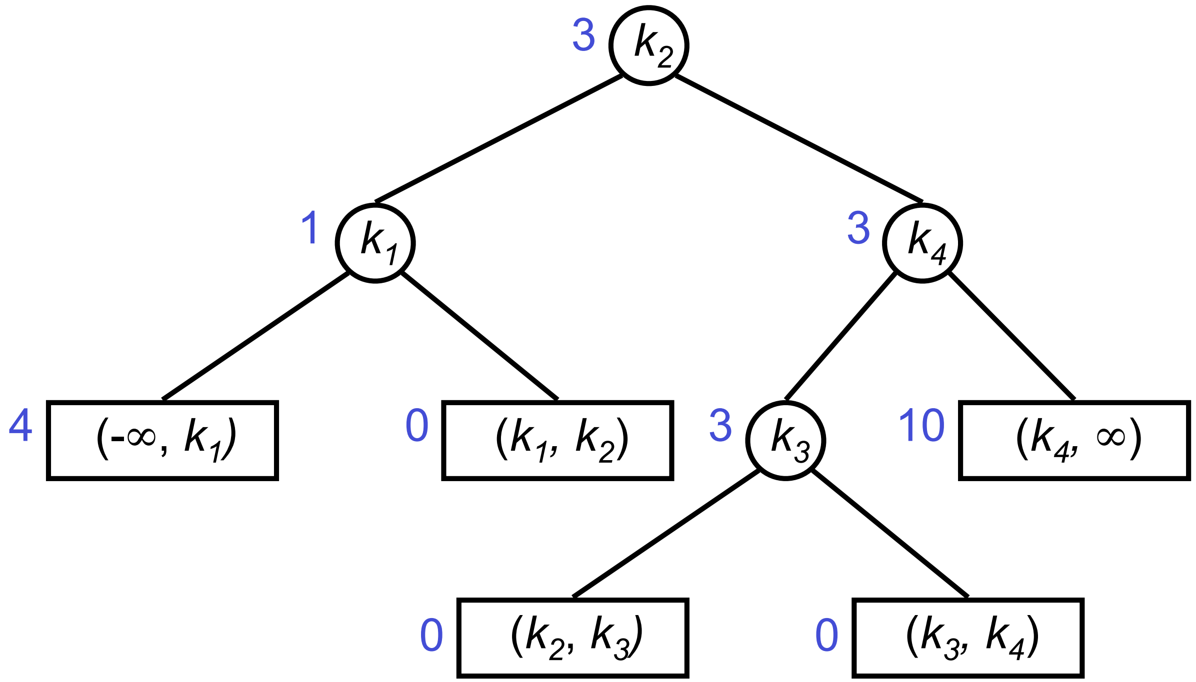
- Optimization problems

An optimal solution to the original problems contains optimal solutions to smaller subproblems

Construction of Optimal Binary Search Tree

$(-\infty,k_1)$ k_1 (k_1,k_2) k_2 (k_2,k_3) k_3 (k_3,k_4) k_4 (k_4, ∞)

4 1 0 3 0 3 0 3 10



weighted path length:

$3 \cdot 1 + 2 \cdot (1 + 3) + 3 \cdot 3 + 2 \cdot (4 + 10)$

Construction of Optimal Binary Search Trees

Give: set of keys *S*

$S = \{k_1, \ldots. k_n\}$ $-\infty = k_0 < k_1 < \ldots < k_n < k_{n+1} = \infty$

a_i: (absolute) frequency of request to key k_i

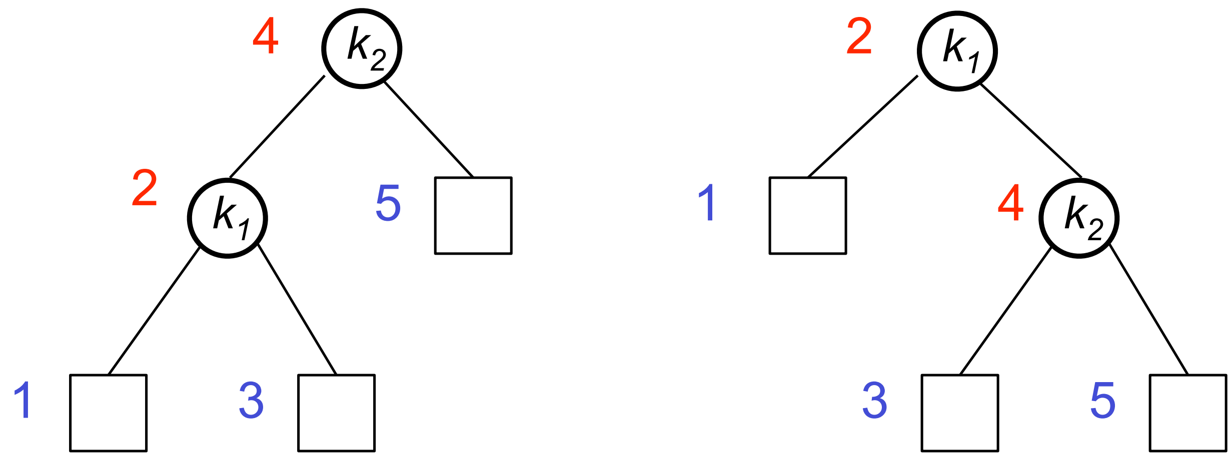
b_j: (absolute) frequency of request to $x \in (k_j, k_{j+1})$

Weighted path length *P*(T) of a binary search tree T for S:

$$P(T) = \sum_{i=1}^{n} (depth(k_i) + 1) a_i + \sum_{j=0}^{n} depth(k_j, k_{j+1}) b_j$$

Goal: Binary search tree with minimum weighted path length P for S

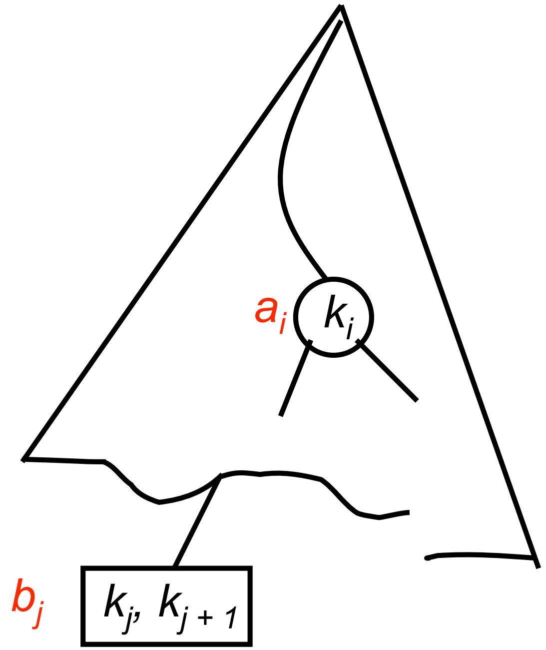
Construction of Optimal Binary Search Trees



$P(T_1) = 21$

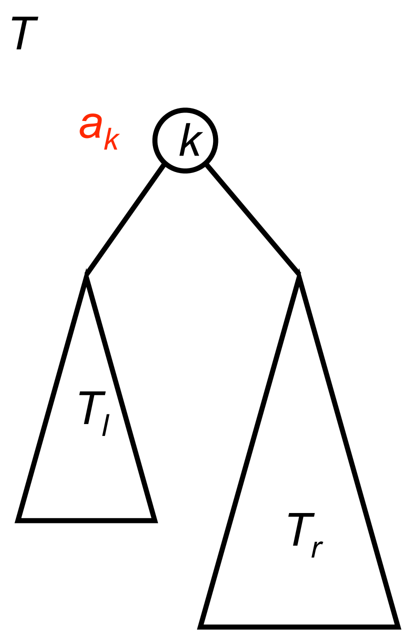
$P(T_2) = 27$

Construction of Optimal Binary Search Trees



An optimal binary search tree is a binary search tree with minimum weighted path length.

Construction of Optimal Binary Search Tree



$$P(T) = P(T_l) + W(T_l) + P(T_r) + W(T_r) + a_{root}$$

$$= P(T_l) + P(T_r) + W(T) \text{ where}$$

$W(T)$:= total weight of all nodes in T

If T is a tree with minimum weighted path length S, then subtree T_l and T_r are trees with minimum weighted path length for subsets of S.

Construction of Optimal Binary Search Trees

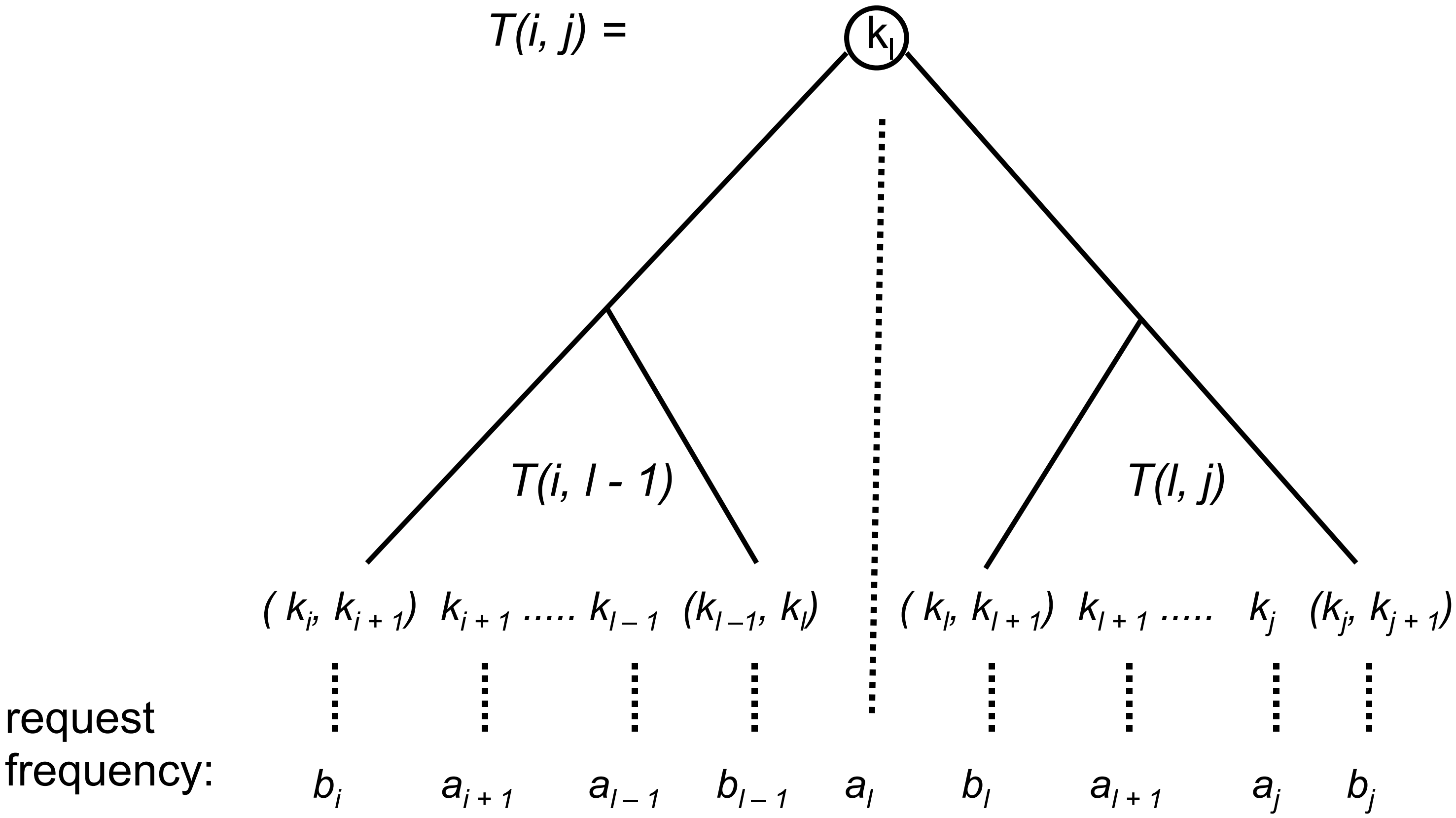
Let

T(i, j): optimal binary search tree for $(k_i, k_{i+1})\ k_{i+1} \ldots k_j\ (k_j, k_{j+1})$,

W(i, j): weight of T(i, j), i.e. $W(i, j) = b_i + a_{i+1} + \ldots + a_j + b_j$,

P(i, j): weighted path length of T(i, j).

Construction of Optimal Binary Search Trees



Construction of Optimal Binary Search Trees

$W(i, i) = b_i$, for $0 \leq i \leq n$

$W(i, j) = W(i, j-1) + a_j + b_j$, for $0 \leq i < j \leq n$

$P(i, i) = 0$, for $0 \leq i \leq n$

$$P(i, j) = W(i, j) + \min_{i < l \leq j} \{ P(i, l-1) + P(l, j) \}, \text{ for } 0 \leq i < j \leq n \quad (*)$$

$r(i, j)$ = the index l for which the minimum is achieved in (*)

Construction of Optimal Binary Search Trees

Base cases

Case 1: $h = j - i = 0$

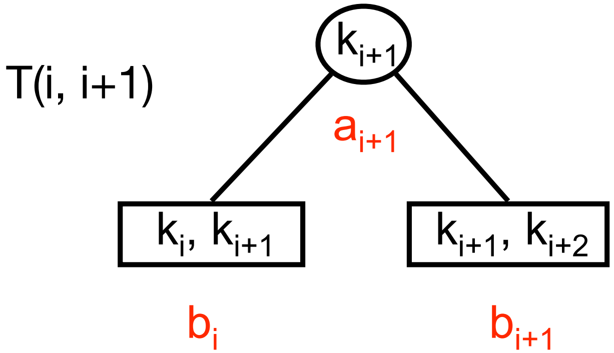
$T(i, i) = (k_i, k_{i+1})$

$W(i, i) = b_i$

$P(i, i) = 0$, $r(i, i)$ not defined

Construction of Optimal Binary Search Trees

Case 2: $h = j - i = 1$

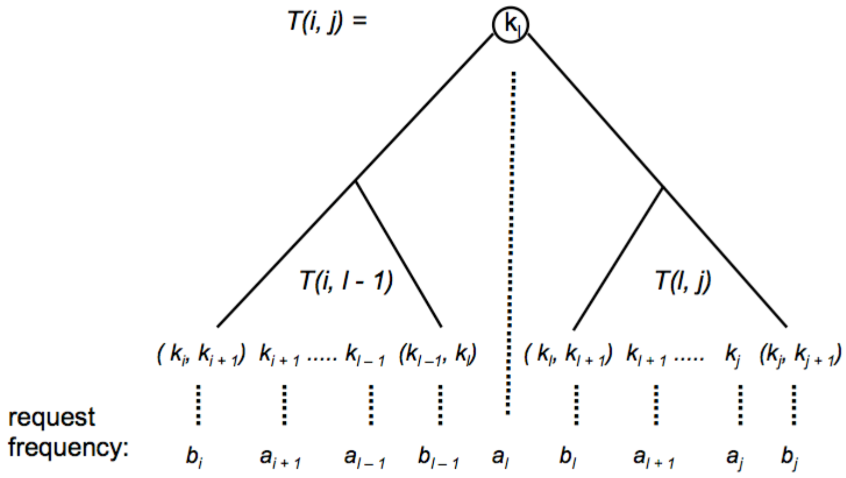


$$W(i, i+1) = b_i + a_{i+1} + b_{i+1} = W(i, i) + a_{i+1} + W(i+1, i+1)$$

$$P(i, i+1) = W(i, i+1)$$

$$r(i, i+1) = i + 1$$

Computing the Minimum Weighted Path Length using Dynamic Programming



Case 3: $h = j - i > 1$

for h = 2 **to** n **do**

for i = 0 **to** (n – h) **do**

{ j = i + h;

determine (largest) l, $i < l \leq j$, s.t. $P(i, l-1) + P(l, j)$ is minimal

$P(i, j) = P(i, l-1) + P(l, j) + W(i, j)$;

$r(i, j) = l$;

}

Construction of Optimal Binary Search Trees

Define:

$$\left.\begin{array}{lcl} P(i,j) & := & \text{minimum weighted path length for} \\ W(i,j) & := & \text{sum of} \end{array}\right\} b_i a_{i+1} b_{i+1} \ldots a_j b_j$$

Then:

$$W(i,j) = \begin{cases} b_i & \text{if } i = j \\ W(i,j-1) + a_j + W(j,j) & \text{otherwise} \end{cases}$$

$$P(i,j) = \begin{cases} 0 & \text{if } i = j \\ W(i,j) + \min_{i<\ell\leq j}\{P(i,\ell-1) + P(\ell,j)\} & \text{otherwise} \end{cases}$$

→ Computing the solution $P(0,n)$ takes time $O(n^3)$ and requires $O(n^2)$ space

Construction of Optimal Binary Search Trees

Theorem

An optimal binary search tree for n keys and $n+1$ intervals with known request frequencies can be constructed in $O(n^3)$ time.

Method of Dynamic Programming

- **Recursive approach**
  - Solve a problem by solving several smaller analogous subproblems of the same type.
  - Then combine these solutions to generate a solution to the original problem.
- **Drawback: Repeated computation of solutions**
- **Dynamic programming**
  - Once a subproblem has been solved, store its solution in a table so that it can be retrieved later by simple table lookup

Dynamic Programming

- **Algorithm design technique, often applied to optimization problems**
- **Generally suitable for recursive approaches, when solution to subproblems are required repeatedly**
- **Approach**
  - maintain a table of subproblem solutions
- **Advantage**
  - improved running time
  - often polynomial instead of exponential

String Matching Problems

Edit Distance

For two given strings A and B, compute the edit distance D(A,B) as well as a minimum sequence of edit operations that transforms A ino B.

```
m a - t h e m - - a t i c i a n
m u l t i p l i c a t i o - - n
```

String Matching Problems

Approximate String Matching

For a given text T, a pattern P and a distance d, find all substrings P' of T with $D(P,P') \leq d$

Sequence Alignment

Find optimal alignments of DNA sequences

```
G A G C A - C T T G G A T T C T C G G
- - - C A C G T G G - - - - - - - - -
```

Edit Distance

Given: Strings A = $a_1 a_2 .... a_m$ and $B = b_1 b_2 ... b_n$

Goal: Minimum number $D(A,B)$ of edit operations required to transform A into B.

Edit operations:

1. Replace a character fom string *A* by a character from *B*
2. Delete a character from string *A*
3. Insert a character from string *B* into string *A*.

```
m a - t h e m - - a t i c i a n
m u l t i p l i c a t i o - - n
```

Edit Distance

Unit cost model:

for a,b being characters or empty words, i.e. ε

$$c(a,b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{if } a = b \end{cases}$$

We want to have a metric. Hence it should satisfy the triangle inequality:

$$c(a,c) \leq c(a,b) + c(b,c)$$

→ for strings only one letter is changed at a time
→ each change increases the cost by one unit

Edit Distance

Trace as representation of the sequence of edit operations:

```
A =     b a a c a a b c
        | |  / /  |  /
B =   a b a c b c a c
```

or using indents

```
A =   - b a a c a - a b c
        | |   | |   |   |
B =   a b a - c b c a - c
```

Edit distance (costs) : 5

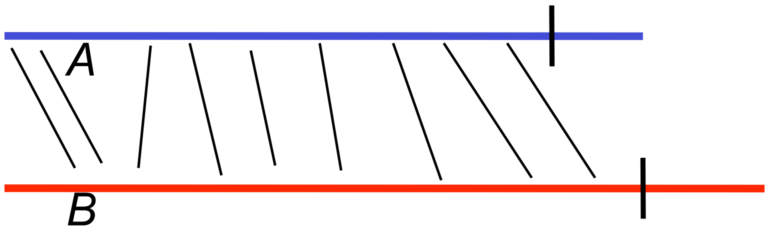
Splitting an optimal trace yields two optimal subtraces

→ dynamic programming is suitables

Computation of the Edit Distance

Let $A_i = a_1 \ldots a_i$ and $B_j = b_1 \ldots. b_j$

$$D_{i,j} = D(A_i, B_j)$$



Computation of the Edit Distance

Three ways of ending a trace

1. a_m is replaced by b_n:

   $D_{m,n} = D_{m-1,n-1} + c(a_m, b_n)$

2. a_m is deleted: $D_{m,n} = D_{m-1,n} + 1$

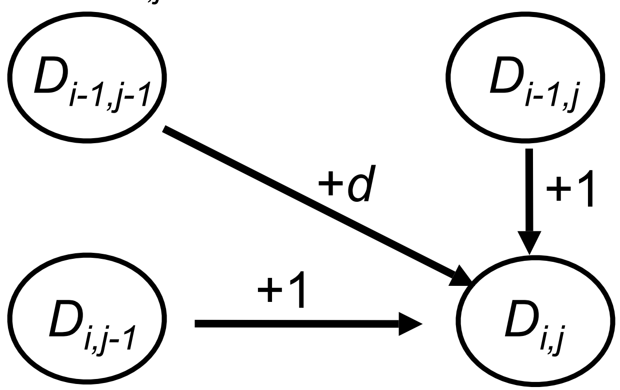
3. b_n is inserted: $D_{m,n} = D_{m,n-1} + 1$

Computation of the Edit Distance

Recurrence relation (*m,n* ≥ 1):

$$D_{m,n} = \min \begin{Bmatrix} D_{m-1,n-1} & + & c(a_m, b_n) \\ D_{m-1,n} & + & 1 \\ D_{m,n-1} & + & 1 \end{Bmatrix}$$

→ computation of all $D_{i,j}$ necessary, $0 \le i \le m$, $0 \le j \le n$.



Recurrences for the Edit Distance

Base case:

$D_{0,0} = D(\varepsilon, \varepsilon) = 0$

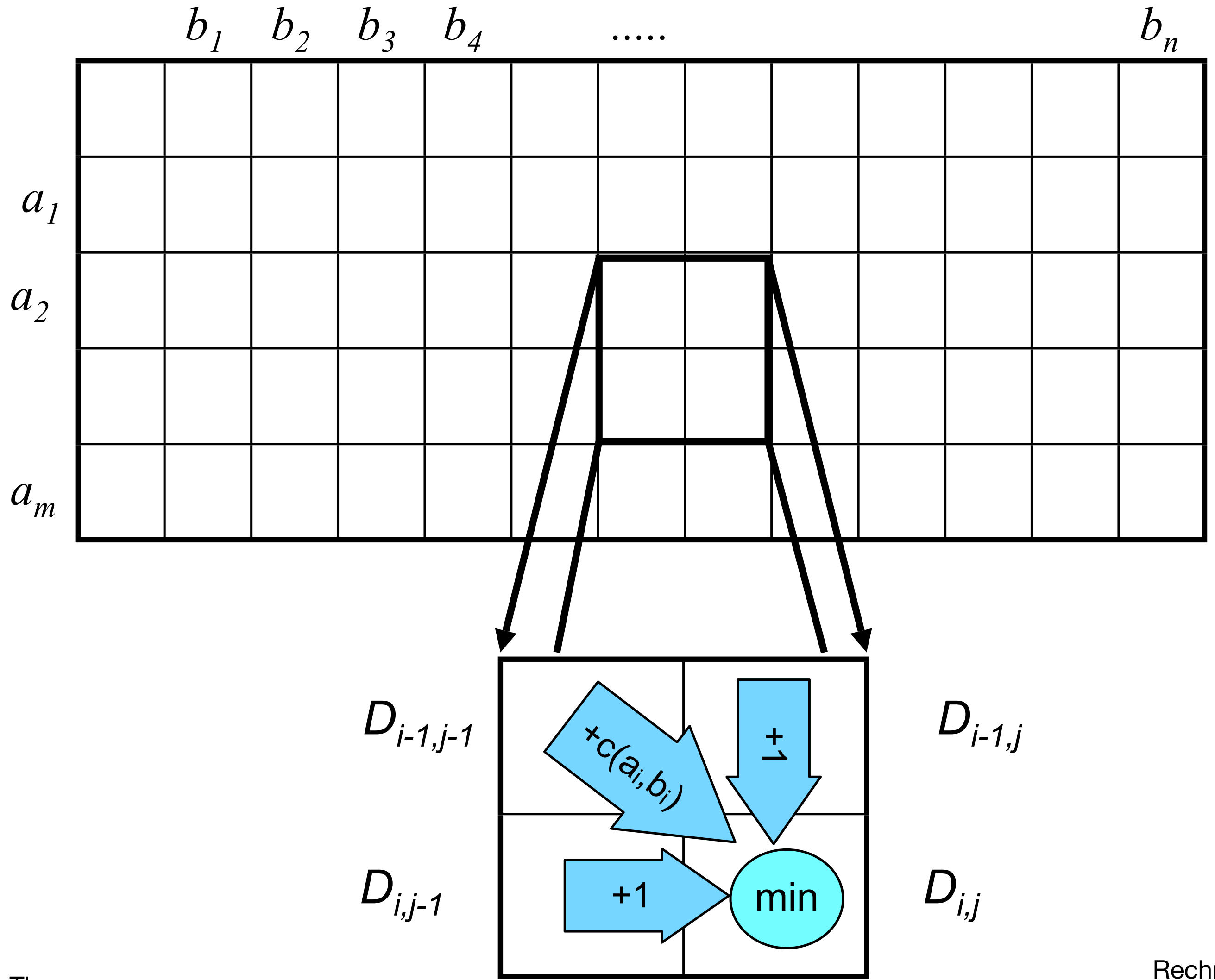
$D_{0,j} = D(\varepsilon, B_j) = j$

$D_{i,0} = D(A_i, \varepsilon) = i$

Recurrence equation:

$$D_{i,j} = \min \begin{Bmatrix} D_{i-1,j-1} & + & c(a_i, b_j) \\ D_{i-1,j} & + & 1 \\ D_{i,j-1} & + & 1 \end{Bmatrix}$$

Order of Solving the Subproblems



Algorithm for Computing the Edit Distance

Algorithm Edit-Distance

Input: Strings $A = a_1 .... a_m$ and $B = b_1 ... b_n$

Output: Matrix $D = (D_{ij})$

$D[0,0] := 0$

for i **:= 1 to** m **do** $D[i,0] = i$

for j **:= 1 to** n **do** $D[0,j] = j$

for i **:= 1 to** m **do**

  for j **:= 1 to** n **do**

    $D[i,j] := \min(D[i-1,j] + 1,$

      $D[i,j-1] + 1,$

      $D[i-1, j-1] + c(a_i, b_j))$

Example

		a	b	a	c
	0	1	2	3	4
b	1	1	1	2	3
a	2	1	2	1	2
a	3	2	2	2	2
c	4	3	3	3	2

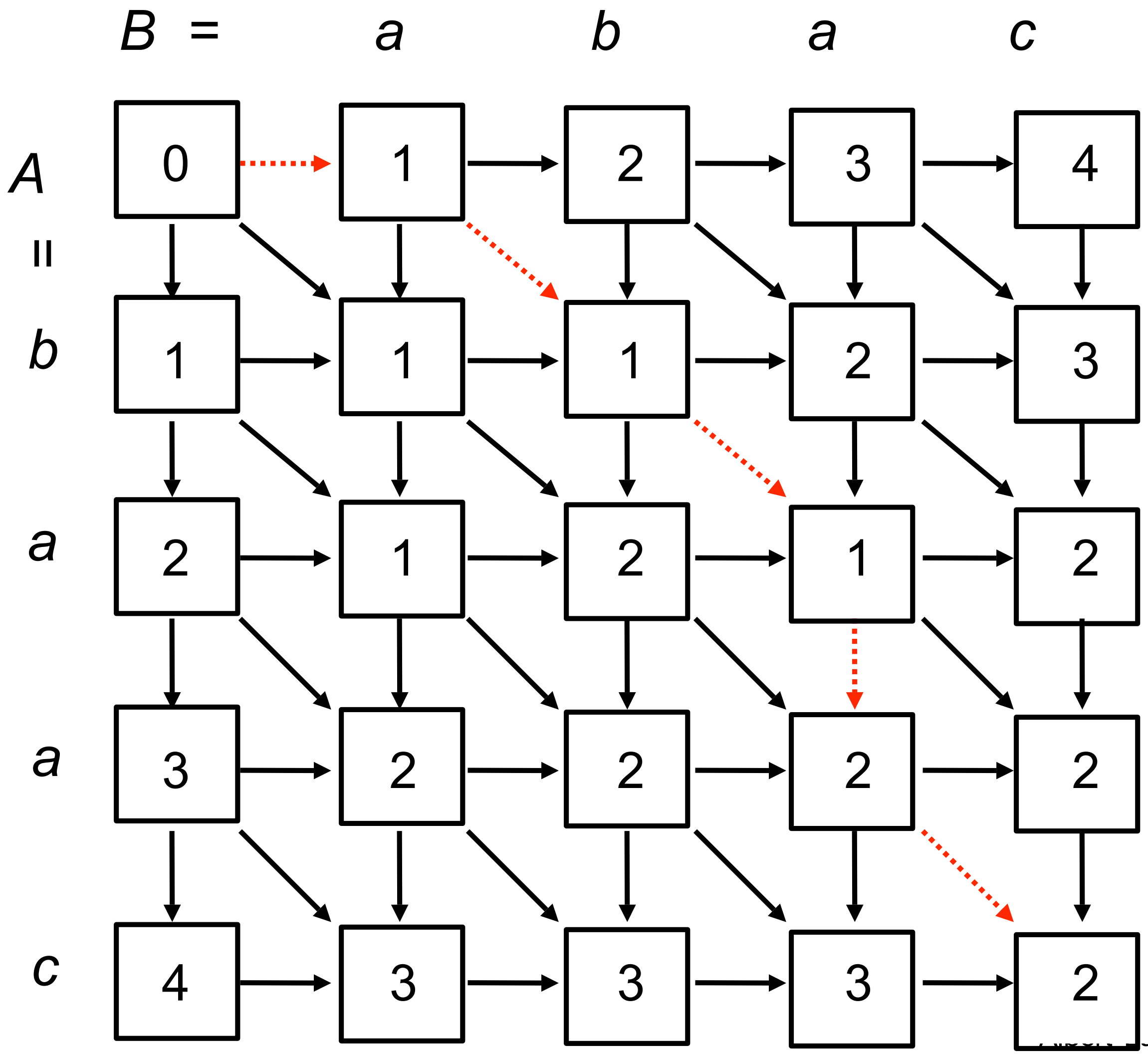
Computing the Edit Operations

Algorithm Edit-operations (*i,j*)
Input: matrix *D* (already computed)
Output: sequence of edit operations

if *i* **= 0 and** *j* **= 0 then return**
if *i* **≠ 0 and** *D*[*i,j*] = *D*[*i* – 1, *j*] + 1
  then Edit-operations (*i* – 1, *j*)
    „delete *a*[*i*]“
else if *j* ≠ 0 and *D*[*i,j*] = *D*[*i*, *j* – 1] + 1
  then Edit-operations(*i,j* – 1)
    „insert *b*[*j*]“
else /* *D[i,j] = D[i – 1, j – 1] + c(a[i], b[j])* */
    Edit-operatoins (*i* – 1, *j* – 1)
    „replace *a[i]* by *b[j]* “

Inital call: Edit-operations(*m,n*)

Trace Graph of Edit Operations



Trace Graph of the Edit Operations

- **Trace Graph:**
  - Representation of a all possible traces of operations that transform A into B. Direct edges from vertex (i,j) to vertices (i+1), (i,j+1) and (i+1,j+1).
- **Edge weights represent the edit costs.**
- **Along an optimal path, costs increase monotonically**
- **Each path from upper left corner to the lower right corner with monotonically increasing costs represents an optimal trace**

Approximate String Matching

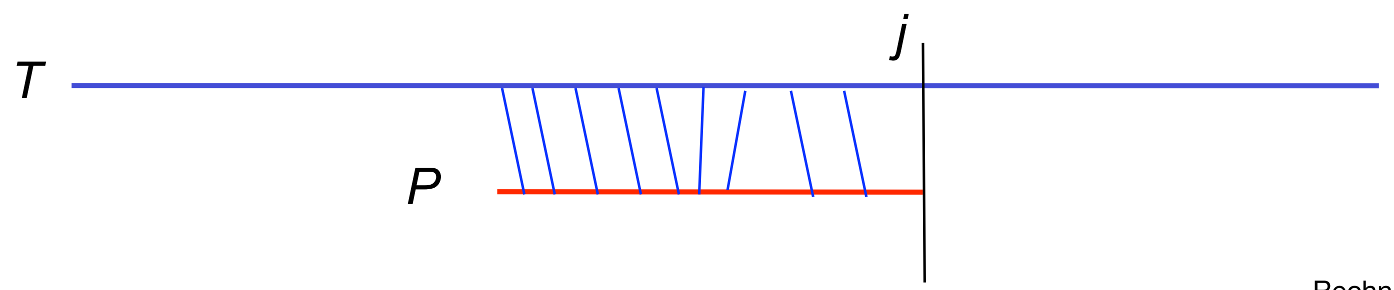
Given: pattern string $P = p_1 p_2 \ldots p_m$ and text string $T = t_1 t_2 \ldots t_n$

Goal: Find an interval $[j', j]$, $1 \leq j', j \leq n$, such that the substring

$T_{j',j} = t_{j'} \ldots t_j$ is the one with the highest similarity to the pattern P.

Thus, for all other intervals $[k', k]$, $1 \leq k', k \leq n$:

$$D(P, T_{j',j}) \leq D(P, T_{k',k})$$



Approximate String Matching

Naive approach:

for all $1 \leq j', j \leq n$ **do**

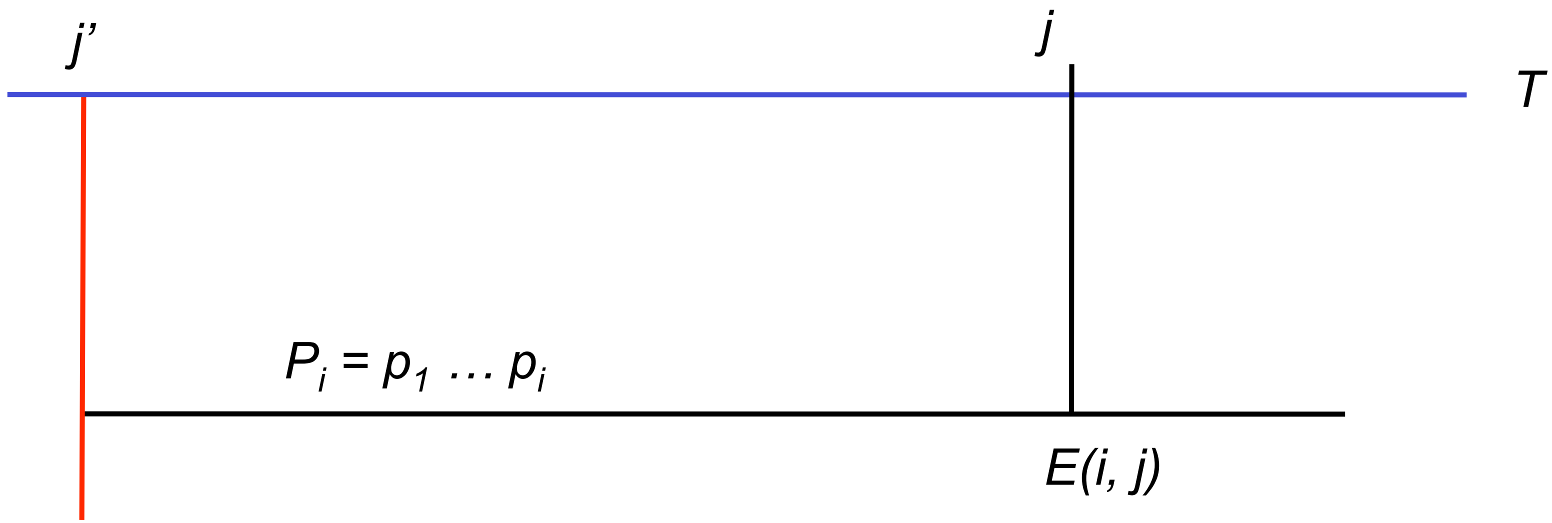
compute $D(P, T_{j', j})$

choose the minimum

Running Time $O(n^3 m)$

Approximate String Matching

Consider a related problem:



For each position j in the text and each position i in the pattern compute the minimum edit distance between P_i and any substring $T_{j',j}$ of T that ends at position j.

Approximative String Matching

Method:

for all $1 \le j \le n$ ***do***

determine j', so that $D(P, T_{j',j})$ is minimized

For $1 \le i \le m$ and $0 \le j \le n$ let:

$$E_{i,j} = \min_{1 \le j' \le j+1} D(P_i, T_{j',j})$$

Optimal trace:

```
P_i    =  b  a  a  c  a  a  b  c
          |  |   /  /     |   /
T_j',j = b  a  c  b  c  a  c
```

Approximative String Matching

Recurrence equation:

$$E_{i,j} = \min \begin{Bmatrix} E_{i-1,j-1} + c(p_i, t_j), \\ E_{i-1,j} + 1, \\ E_{i,j-1} + 1 \end{Bmatrix}$$

Remarks:

The index *j´* may differ for $E_{i-1, j-1}$, $E_{i-1,j}$ and $E_{i, j-1}$

A subtrace of an optimal trace is an optimal subtrace.

Approximate String Matching

Base case:

$$E_{0,0} = E(\varepsilon, \varepsilon) = 0$$

$$E_{i,0} = E(P_i, \varepsilon) = i$$

whereas

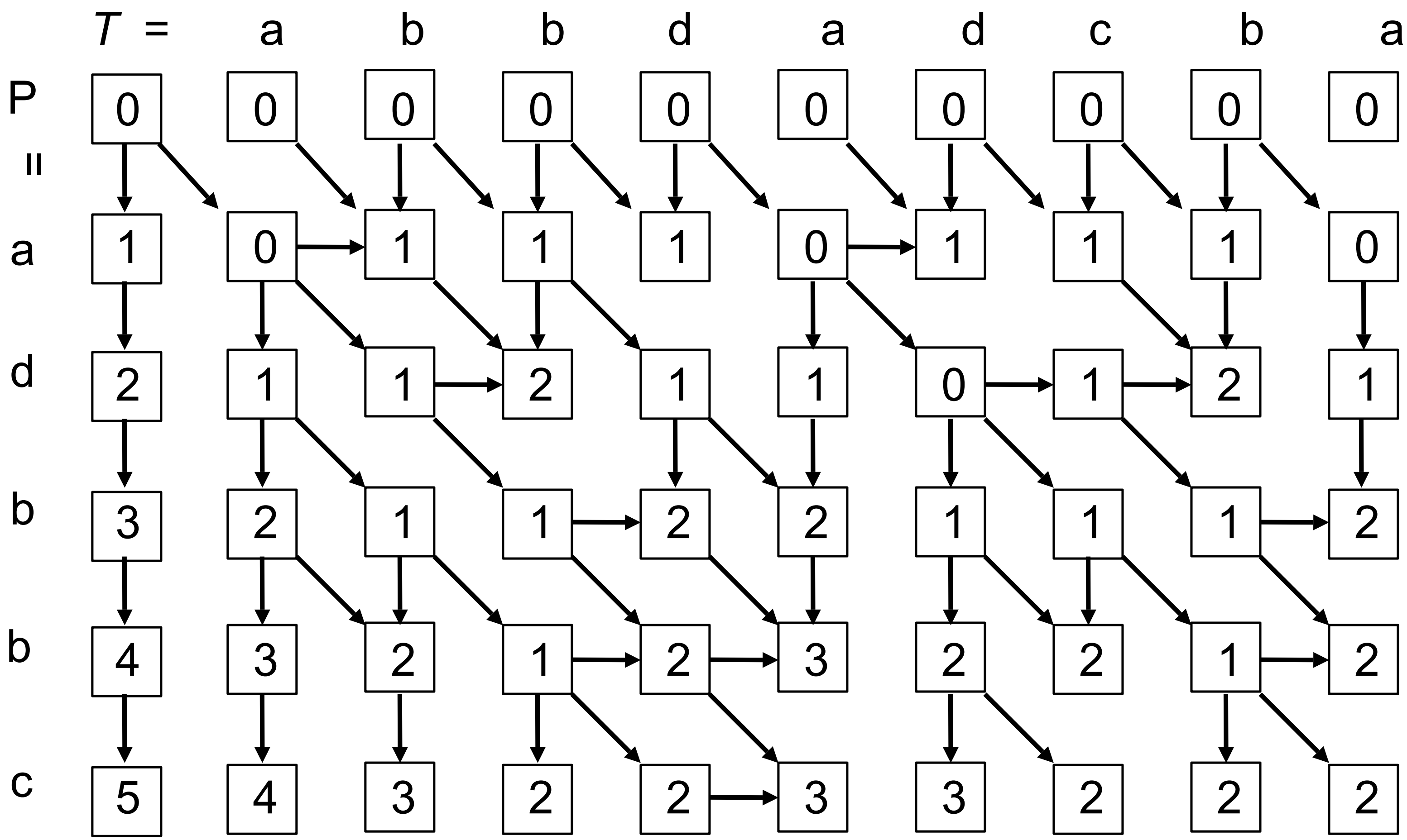
$$E_{0,j} = E(\varepsilon, T_j) = 0$$

Observation:

An optimal sequence of edit operations that transforms P into $T_{j', j}$ does not start with an insertion of character $t_{j'}$.

Approximate String Matchings

Dependency Graph



Approximate String Matching

Theorem

If there is a path from $E_{0, j'-1}$ to $E_{i, j}$, in the dependency graph, them $T_{j', j}$ is a substring of T that has the highest to P_i, ending at position j and satisfying

$$D(P_i, T_{j', j}) = E_{i, j}$$

Similarity of Strings

Sequence Alignment:

For two given DNA sequences, insert spaces (or dashes) such that after placing the resulting strings one above the other, the number of matching characters is maximized.

```
G A - C G G A T T A G
G A T C G G A A T A G
```

Similarity of Strings

Similarity measure for characters

example	setting	in general
+ 1	for a match	} *s*(a,b)
- 1	for a mismatch	
- 2	for spaces	- c

Measuring the similarity of two sequences

$$S(A,B) = \sum_{\text{characters } a_i, b_i} \text{similarity of}(a_i, b_i)$$

Goal: Find alignment optimizing similarity

Similarity of Strings

Similarity *S(A,B)* of two strings *A* and *B*

Operations:

1. Replacement of a character a by some character b: $s(a,b)$
2. Deletion of a character from *A*, insertion of a character from B, Loss: $-c$

Goal:

Find a sequence of operations that transforms *A* into *B* such that the total gain is maximized.

Similarity of Strings

$S_{i,j} = S(A_i, B_j)$, $0 \le i \le m$, $0 \le j \le n$

Recurrence equation:

$$S_{m,n} = \max \; (S_{m-1,n-1} + s(a_m, b_n), \; S_{m-1,n} - c, \; S_{m,n-1} - c)$$

Initial condition:

$$S_{0,0} = S(\varepsilon, \varepsilon) = 0$$

$$S_{0,j} = S(\varepsilon, B_j) = -jc$$

$$S_{i,0} = S(A_i, \varepsilon) = -ic$$

Most Similar Substring

Given: Two strings $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_n$

Goal: Find two intervals $[i', i] \subseteq [1, m]$ and $[j', j] \subseteq [1, n]$ with

$$S(A_{i',i}, B_{j',j}) \geq S(A_{k',k}, B_{l',l}),$$

for all $[k',k] \subseteq [1, m]$ and $[l',l] \subseteq [1, n]$.

Naive Approach:

for all $[i', i] \subseteq [1, m]$ **and** $[j', j] \subseteq [1, n]$ **do**

compute $S(A_{i',i}, B_{j',j})$

Running time: $O(m^2n^2)$

Most Similar Substrings

Method:

for all $1 \leq i \leq m$, $1 \leq j \leq n$ **do**

Compute i' und j', such that $S(A_{i',i}, B_{j',j})$ is maximal

For $0 \leq i \leq m$ und $0 \leq j \leq n$ let:

$$H_{i,j} = \max_{\substack{1 \leq i' \leq i+1, \\ 1 \leq j' \leq j+1}} S(A_{i',i}, B_{j',j})$$

Optimal trace

$$\begin{array}{rccccccccc} A_{i',i} = & b & a & a & c & a & - & a & b & c \\ & | & | & & | & | & & | & & | \\ B_{j',j} = & b & a & - & c & b & c & a & - & c \end{array}$$

Most Similar Substring

Recurrence relation:

$$H_{i,j} = \max \begin{Bmatrix} H_{i-1,j-1} + s(a_i, b_j) \\ H_{i-1,j} - c \\ H_{i,j-1} - c \\ 0 \end{Bmatrix}$$

in our example:
s(a,a) = +1
s(a,b) = -1 for a≠b
c = -2 (inserting/deleting)

Base cases:

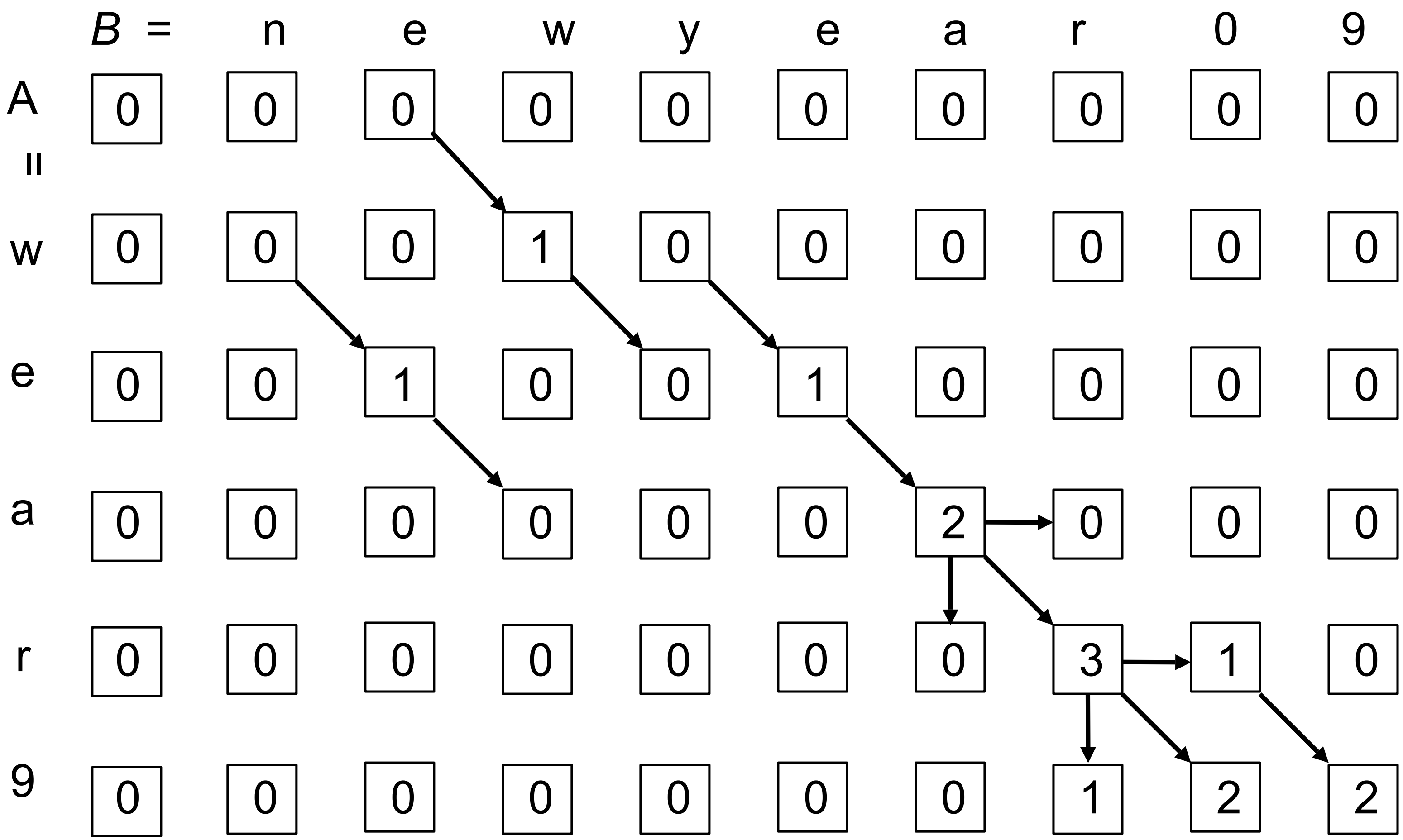
$$H_{0,0} = H(\varepsilon, \varepsilon) = 0$$

$$H_{i,0} = H(A_i, \varepsilon) = 0$$

$$H_{0,j} = H(\varepsilon, B_j) = 0$$

Most similar substring

Dependency Graph



ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Algorithm Theory

11 Dynamic Programming

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08

CoNe Freiburg

IIF
INSTITUT FÜR
INFORMATIK
FREIBURG