## Inhalte der Vorlesung

### Themen

- Dynamisches Programmieren
- paarweises und multiples Sequenzalignment
- Bio-Datenbanken, Homologiesuche
- Sequenzierung und Genomassemblierung
- Hidden-Markow-Modelle
- Genvorhersage
- Genomdarstellung / Genombrowser
- Phylogenie
- Proteinfamilien
References

References for this lecture

- WIKIPEDIA
- OR/MS Games: 4. The Joy of Egg-Dropping in Braunschweig and Hong Kong,
Egg dropping puzzle

Example (Egg dropping puzzle)

Objective: We want to know from which stories of a 21-story building it is safe to drop eggs from. (chalk board)

Assumptions:
- The effect of a fall is the same for all eggs (experiment repeatable)
- If an egg breaks when dropped, then it would break if dropped from a higher window (monotonicity)
- A broken egg must be discarded
- An egg that survives a fall can be used again
- It is possible that eggs break when dropped from first floor and that they do not break when thrown from the 21th floor

Problem: Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?
## Egg dropping puzzle (general)

We have $N$ eggs and the building has $S$ stories ($N = 2$, $S = 21$ in example). Let $M \in 0, 1, 2, \ldots, S$ be the largest floor where the egg survives ($M$ unknown).

### Observations:

- After any number of egg-droppings the set of still possible values of $M$ is a consecutive interval of, say $s$, floors.
- The maximal number of remaining egg-dropping depends only on $s$ and the number of remaining eggs.

Define for $n = 1, \ldots, N$ and $s = 1, \ldots, S$

<table>
<thead>
<tr>
<th>$W(n, s)$</th>
<th>minimum number of trials required to identify the value of the critical floor $M$ under the worst case scenario given that $n$ eggs are available and that $s$ floors are yet to be tested</th>
</tr>
</thead>
</table>

$W(N, S) =$?
Egg dropping puzzle

Boundary Cases

- \(W(1, s) = s\) for all \(s = 1, 2, \ldots\)
  If there is only one egg we need \(s\) tries to find the right floor in the worst case: Start with egg-dropping from first floor. Then; if not broken; drop from second floor, etc.

- \(W(n, 1) = 1\) for all \(n = 1, 2, \ldots\)
  If there is only one floor left that needs to be tested then we need 1 try.

- \(W(n, 0) := 0\) for all \(n = 1, 2, \ldots\) (defined for convenience)
  If there is no floor left we need no more tries.
**Egg dropping recursion**

**Theorem**

The recursion equation

\[ W(n, s) = 1 + \min_{\ell=1,2,...,s} \left\{ \max\{ W(n-1, \ell-1), W(n, s-\ell) \} \right\} \]

holds for \( n = 2, ..., N \), \( s = 2, 3, 4, ..., H \).
Egg dropping recursion

Proof.

Suppose \( n \) eggs are left and \( s \) floors yet to be tested, wlg the floors \( 1, 2, \ldots, s \).
Let \( k \) be the first tested floor in an optimal strategy that is guaranteed to find the right answer.
Let \( W(n, s, \ell) \) be the number of tries in the worst case of an optimal strategy that starts with testing the \( \ell \)th floor when \( n \) eggs and \( s \) floors left.
We then have
\[
W(n, s) = W(n, s, k)
\]
by the choice of \( k \). We also have
\[
W(n, s) = \min_{\ell = 1, 2, \ldots, s} W(n, s, \ell).
\] (1)

Here, “\( \geq \)” holds trivially and “\( \leq \)” holds because otherwise choosing the \( k \)th floor in the first drop would not be optimal.
Now distinguish two cases

a) The egg breaks when thrown from \( \ell \)th floor.
   Then \( W(n, s, \ell) = 1 + W(n - 1, \ell - 1) \) by definition of \( W(\cdot, \cdot) \), because we need the first try plus the optimal number of tries being left with \( n - 1 \) eggs and the first \( \ell - 1 \) floors yet to be tested.

b) The egg survives when thrown from \( \ell \)th floor.
   Then \( W(n, s, \ell) = 1 + W(n, s - \ell) \), because we need the first try plus the optimal number of tries with \( n \) eggs and the upper \( s - \ell \) floors yet to be tested.

As we are considering the worst case we have
\[
W(n, s, \ell) = \max \{1 + W(n - 1, \ell - 1), 1 + W(n, s - \ell)\} = 1 + \max \{W(n - 1, \ell - 1), W(n, s - \ell)\}
\] (2)

Plugging (2) in (1) yields the claimed recursion. \( \square \)
Iteratively compute $W(n, s)$

Example: $N = 3$, $S = 10$ (3 eggs, 10 stories)

1) Use a table for $W(n, s)$, where $n = 1, 2, \ldots, N$, $s = 0, 1, \ldots, S$
2) Fill in boundary cases
3) Use recursion

$$W(n, s) = 1 + \min_{\ell=1, 2, \ldots, s} \{ \max\{ W(n - 1, \ell - 1), W(n, s - \ell) \} \}$$

to iteratively fill in table.

<table>
<thead>
<tr>
<th>$W(n, s)$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Choose an order so that values on the rhs are already computed when entry $(n, s)$ is computed. E.g.

for $(n=2; n<=N; n++)$

for $(s=2; s<=S; s++)$

compute $W(n, s)$
Iteratively compute $W(n, s)$

Example: $N = 3$, $S = 10$ (3 eggs, 10 stories)

1) Use a table for $W(n, s)$, where $n = 1, 2, \ldots, N$, $s = 0, 1, \ldots, S$
2) Fill in boundary cases
3) Use recursion

$$W(n, s) = 1 + \min_{\ell=1,2,\ldots,s} \{ \max \{ W(n-1, \ell-1), W(n, s-\ell) \} \}$$

to iteratively fill in table.

<table>
<thead>
<tr>
<th>$W(n, s)$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$W(2, 2) = 1 + \min \{ \max \{ W(1, 0), W(2, 1) \}, \max \{ W(1, 1), W(2, 0) \} \}$
Iteratively compute \( W(n, s) \)

Example: \( N = 3, S = 10 \) (3 eggs, 10 stories)

1) Use a table for \( W(n, s) \), where \( n = 1, 2, \ldots, N \), \( s = 0, 1, \ldots, S \)
2) Fill in boundary cases
3) Use recursion

\[
W(n, s) = 1 + \min_{\ell=1,2,\ldots,s} \{ \max\{W(n-1, \ell-1), W(n, s-\ell)\} \}
\]

to iteratively fill in table.

\[
\begin{array}{llllllllll}
W(n, s) & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
n & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
& 2 & 0 & 1 & 2 & 2 & & & & & & & \\
& 3 & 0 & 1 & & & & & & & & & \\
\end{array}
\]

\[
W(2, 3) = 1 + \min\{\max\{W(1, 0), W(2, 2)\}, \max\{W(1, 1), W(2, 1)\}, \max\{W(1, 2), W(2, 0)\}\}
\]
Iteratively compute $W(n, s)$

Example: $N = 3$, $S = 10$ (3 eggs, 10 stories)

1) Use a table for $W(n, s)$, where $n = 1, 2, \ldots, N$, $s = 0, 1, \ldots, S$
2) Fill in boundary cases
3) Use recursion

$$W(n, s) = 1 + \min_{\ell=1,2,\ldots,s} \{ \max \{ W(n-1, \ell-1), W(n, s-\ell) \} \}$$

to iteratively fill in table.

<table>
<thead>
<tr>
<th>$W(n, s)$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$W(2, 4) = 1 + \min \{ \max \{ W(1, 0), W(2, 3) \} \}$

$\max \{ W(1, 1), W(2, 2) \}$

$\max \{ W(1, 2), W(2, 1) \}$

$\max \{ W(1, 3), W(2, 0) \}$
Iteratively compute $W(n, s)$

Example: $N = 3, S = 10$ (3 eggs, 10 stories)

1) Use a table for $W(n, s)$, where $n = 1, 2, \ldots, N$, $s = 0, 1, \ldots, S$
2) Fill in boundary cases
3) Use recursion

$$W(n, s) = 1 + \min_{\ell=1,2,\ldots,s} \{\max\{W(n-1, \ell-1), W(n, s-\ell)\}\}$$

to iteratively fill in table.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W(n, s)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$n$</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

last step: $W(3, 10) = 1 + \min\{\max\{W(2, 0), W(3, 9)\}\}$

\[
\max\{W(2, 1), W(3, 8)\} \quad \ldots
\]

\[
\max\{W(2, 8), W(3, 1)\}
\]

\[
\max\{W(2, 9), W(3, 0)\}
\]
Iteratively compute $W(n, s)$

Example: $N = 3, S = 10$ (3 eggs, 10 stories)

1) Use a table for $W(n, s)$, where $n = 1, 2, \ldots, N$, $s = 0, 1, \ldots, S$
2) Fill in boundary cases
3) Use recursion

$$W(n, s) = 1 + \min_{\ell=1,2,\ldots,s} \{ \max\{W(n-1, \ell-1), W(n, s-\ell)\} \}$$

to iteratively fill in table.

4) Find solution $W(N, S) = 4$:
At least 4 egg-droppings are necessary in the worst case.
Solution for 2 eggs and 21 stories

\[ N = 2 \text{ eggs}, \ S = 21 \text{ stories} \]

\[ M(2, 21) = 6 \]

In the worst case 6 egg-droppings are necessary to find the solution for a 21 stories building using 2 eggs.

\[
\begin{array}{cccccccccccccccccc}
\hline
\hline
M(n, s) & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 \\
\hline
k & \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0} &  \phantom{0}
\end{array}
\]
Dynamic Programming

Dynamic Programming (DP)

- is typically applied to optimization problems
- solves problems by combining the solutions to subproblems
  *Optimal substructure* property: An optimal solution to the problem contains optimal solutions to subproblems.
- applicable when the subproblems share subsubproblems (overlapping subproblems property) *(chalk board)*
  If not: a *divide-and-conquer* algorithm may work.
- Computes the value of an optimal solution first.
  Optionally, the optimal solution can be constructed from computed information *(backtracking)*.
Increasing subsequence

**Definition (increasing subsequence)**

Let \( a = (a_1, a_2, \ldots, a_n) \) be a sequence of numbers. \( b = (b_1, b_2, \ldots, b_k) \) is called a subsequence of \( a \) of length \( k \) if indices \( 1 \leq i_1 < i_2 < \ldots < i_k \leq n \) exist such that \( b_j = a_{i_j} \) for \( j = 1, \ldots, k \).

\( b \) is called an increasing sequence if \( b_1 < b_2 < \cdots < b_k \).

**Example**

\[ a = (6, 8, 0, 4, 15, 3, 7, 9, 20, 12, 43, 6, 4, 23) \]

\((8, 9, 20, 23)\) is an increasing subsequence of \( a \) of length 4.
Longest increasing subsequence problem

For a given sequence \( a = (a_1, a_2, \ldots, a_n) \) find a longest increasing subsequence.

An efficient algorithm can be designed using DP.
Longest increasing subsequence problem

**Choose subproblems**

- define DP variables for solutions of subproblems
- must be able to find recursion (optimal substructure property)
- solution to original problem instance should be given by DP variables
Ideas for subproblems

Example (prefixes as subproblems)

$L(k) := \text{length of longest increasing subsequence of } (a_1, \ldots, a_k) \text{ for } k = 1, 2, \ldots, n.$

does not work

Example (prefixes with constraint as subproblems)

$L(k) := \text{length of longest increasing subsequence of } (a_1, \ldots, a_k) \text{ that ends in } a_k \text{ for } k = 1, 2, \ldots, n.$

good

Example (subintervals with constraints as subproblems)

$L(k, \ell) := \text{length of longest increasing subsequence of } (a_k, \ldots, a_\ell) \text{ that starts with } a_k \text{ and ends with } a_\ell \text{ for } 1 \leq k \leq \ell \leq n.$

unnecessarily complicated

(chalk board)
### Find Recursion

#### Definition (DP variables)

\[ L(k) := \text{length of longest increasing subsequence of } (a_1, \ldots, a_k) \]

*that ends in* \( a_k \)

*for* \( k = 1, 2, \ldots, n. \)

#### Find DP recursion

1. **Use intuition and examples to identify solutions to subproblems given a solution to original problem** (*chalk board*)
2. **Construct recursion hypothesis**

\[
L(k) = 1 + \max_{1 \leq i < k} \left\{ L(i) \right\} \quad (3)
\]

*for* \( k = 2, \ldots, n. \) Here, let \( \max\{\} := 0. \)
3. **Define boundary cases:** \( L(1) = 1 \)
4. **Prove recursion**
Proof of Recursion

Proof.

Let \( k \in \{1, 2, \ldots, n\} \).
Let \( b = (a_{j_1}, a_{j_2}, \ldots, a_{j_{m-1}}, a_k) \) be a longest increasing subsequence of \((a_1, \ldots, a_k)\) of length \( m \) that ends in \( a_k \).
In the case \( m = 1 \), we have \( b = (a_k) \) and the recursion is trivially true. Therefore, assume that \( m > 1 \).
By definition of \( L \), we have
\[
L(k) = m.
\]
The sequence \((a_{j_1}, a_{j_2}, \ldots, a_{j_{m-1}})\) (of length \( m-1 \)) must be a longest increasing subsequence of \((a_1, \ldots, a_{j_{m-1}})\), that ends in \( a_{j_{m-1}} \). Otherwise, \( b \) would not be optimal as assumed. Therefore, we get by definition of \( L \) that \( L(j_{m-1}) = m - 1 \). We summarize
\[
L(k) = m = 1 + (m - 1) = 1 + L(j_{m-1}). \tag{4}
\]
Now we show that
\[
L(j_{m-1}) = \max_{1 \leq i < k, a_i < a_k} L(i). \tag{5}
\]
“\( \leq \)”: is trivial as \( j_{m-1} < k \) and \( a_{j_{m-1}} < a_k \) and a maximum is at east as large as any element that is maximized over.
“\( \geq \)”: Assume for the sake of contradiction we had the inequality “\( < \)” in (5). We could then construct an increasing subsequence of \((a_1, \ldots, a_k)\) that ends in \( a_k \) that is longer than \( b \) by using the solution of the \( i \) that maximizes the right hand size of (5). This contradicts the choice of \( b \) and we must have “\( \geq \)” in (5).
Plugging in (5) into (4) yields the claim. \qed
Save backtracking variables for recovery of optimal solution

- In this example, we are interested not only in its length but also in the *actual sequence of the solution*.
- Recovery of optimal solution is done later but simplified by remembering the “choices” made during recursion. Here: \( p(k) = \text{some } i \) with which the maximum in the DP recursion (3) is achieved.
DP Algorithm

DP algorithm to find a longest increasing subsequence

1: \( L(1) \leftarrow 1 \) // initialization of boundary cases
2: \( p(1) \leftarrow 0 \) // initialization of backtracking variables
3: \textbf{for} \( k = 2 \) to \( n \) \textbf{do}
4: \( L(k) \leftarrow 1; \ p(k) \leftarrow 0 \)
5: \textbf{for} \( i = 1 \) to \( k - 1 \) \textbf{do}
6: \quad \textbf{if} \ a_i < a_k \text{ and } 1 + L(i) > L(k) \textbf{then}
7: \quad \quad L(k) \leftarrow 1 + L(i)
8: \quad \quad p(k) \leftarrow i
9: \quad \textbf{end if}
10: \textbf{end for}
11: \textbf{end for}
12: k \leftarrow \arg\max\{L(i) \mid i = 1, \ldots, n\}
13: \text{output } L(k) \text{ as the value of the optimal solution}
14: // backtracking starts
15: b \leftarrow <\text{empty list}> \ // b \text{ will hold optimal solution}
16: \textbf{repeat}
17: \quad \text{add } k \text{ to front of } b\n18: \quad k \leftarrow p(k)
19: \textbf{until} \ k = 0
20: \text{output } b \text{ as optimal solution}
Efficiency of above algorithm

The running time is dominated by the two loops in lines 3 and 5. The inner commands in lines 6-8 take constant time and are executed

\[ \sum_{k=2}^{n} (k - 1) = \frac{n(n - 1)}{2} = O(n^2) \]

times.

The required memory is dominated by the space required for vectors \( L \) and \( p \), each of size \( n \).

**Time:** \( O(n^2) \)

**Space:** \( O(n) \)
Saving Space

Remark: There exist techniques that allow to trade off an increase in running time against a decrease in space. E.g. in the LIS-Problem we could have used an algorithm that takes constant space but twice the running time.