INF 431
Dynamic Programming
PC – April 2013
Weighted interval selection problem

- Assume a set of \{1...n\} time intervals.
- 2 intervals are compatible if they do not overlap
- Let \(p(j)\) is the right most interval ending before interval \(j\) starts
- Each interval has a duration \(u_i\)
- **Problem:** Select the subset \(S\) of \{1,....n\} such that they are all compatible and \(\sum_{i \in S} u_i\) is maximal.
Weighted interval selection problem

• Assume a solution $O$
  – If the last interval $n$ is in $O$
    • No interval between $p(n)$ and $n$ can be in the solution (why?)
    • $O$ must include an optimal solution $\{1, \ldots, p(n)\}$
  – Else if $n$ is not in the solution $O$ then $O = O(n-1)$
  – Thus for each $j$ in $\{1 \ldots n\}$: $O_j$ denotes the optimal solution to the problem $\{1 \ldots j\}$ and $OPT(j)$ is the value of the solution. Thus:

If $j$ in $O$ then $OPT(j) = u_i + OPT(p(j))$
else $OPT(j) = OPT(j-1)$

Therefore: $OPT(j) = \max(u_i + OPT(p(j)), OPT(j-1))$
Weighted interval selection problem

\( j \) belongs to an optimal solution on the set \( \{1,2,...j\} \) iff:

- \( u(j) + OPT(p(j)) \geq OPT(j-1) \)
- Thus dynamic programming essence:
  - A recurrence relation representing the optimal solution (or its value) in terms of optimal solutions to smaller sub problems

Recursive solution:

\[
\text{Compute}_{-}\text{opt}(j) \\
\text{if } j = 0 \text{ return } 0 \\
\text{else return max}(u_j + OPT(p(j)), \text{compute}_{-}\text{opt}(j-1))
\]

Very high complexity..why?
Find a solution to the problem - memoization

- Storing values that are already computed
- Iterative-Compute-Opt
  
  \[ M[0] = 0 \]
  
  for \( j = 1, 2, \ldots, n \)
  
  \[ M[j] = \max(u_j + M[p(j)], M(j-1)) \]

- Complexity?
- \( O(n) \) assuming the intervals are sorted by their ending times
A. The Knapsack problem.. The problem of backpack

Assume a backpack and n objects that we want to carry. The objects are numbered from 1 to n. Each object i has a weight \( p_i \) and a value \( u_i \) which are non-negative integers.

The capacity of the bag, i.e. is to say the maximum weight of its contents, is \( P \).

We search for a subset \( I \) of the objects \( \{1,...,n\} \) to fill the bag

- maximizing the quantity \( \sum_{i \in I} u_i \)
- while respecting the constraint \( \sum_{i \in I} p_i \leq P \)
Q1

• Write the pseudo code of a dynamic programming that computes the optimal value of $\sum_{i \in I} u_i$, without explicitly computing a set $I$ that achieves this optimum value. The complexity must be polynomial in $P$ and $n$. 
Q1 - solution

• Let $c_{i,p}$ the optimal value of the objects 1 to $i$, whose weights do not exceed $p$.

• If object $i+1$ does not appear in the optimal solution for the set $\{1..., i+1\}$ then $c_{i+1,p} = c_{i,p}$

• else $p_{i+1}$ must be reserved for the backpack adding the value: $c_{i+1,p} = u_{i+1,p} + c_{i,p-p_{i+1}}$

• We compute the matrix $c_{i,p}$ for $0<=i<=n$, $0<=p<=P$
Q1 - solution

procédure KNAPSACK(u; p; n; P)
For j = 1 to P
    c[0, j] := 0;
For i = 1 to n
    for j = 1 to P
        c[i, j] := max(c[i -1, j], c[i-1, j-p[i]] + u[i]);
return c[n,P]

The two loops result in a complexity \( O(nP) \).
Question 2 Find a subset \( I \) that reached this optimum value – in the same complexity

- Once the table of the previous example is filled it suffices that we go to \( cn, P \) to find the choices made.

- The following procedure constructs an optimal solution, encoded by a Boolean array.

```plaintext
procédure SELECTION (u, p, c, n, P)
for (i = n, j = P; i>0; i--;
    Sol[i]:= (c[i,j] != c[i-1,j]) ; //Sol = boolean variable
if Sol[i]
    then j = j – p(i);
```
B. Longest common sequence

• Assume the two strings: AGTTACG and CGATAAG. We are searching for the maximal k sequence (not necessarily adjacent letters) in the two strings (i.e.: ATAG)

• **Case1: two sequences both end in the same element.** To find their LCS:
  – shorten each sequence by removing the last element,
  – find the LCS of the shortened sequences,
  – to that LCS append the removed element.

• example, *(BANANA) and (ATANA)*
  – Remove the same last element.
  – Repeat the procedure until you find no common last element –removed sequence will be (ANA).
  – The sequences now under consideration: (BAN) and (AT)
  – The LCS of these last two sequences is, by inspection, (A).
  – Append the removed element, (ANA), giving *(AANA)*: the LCS of the original sequences.

• **Thus** $LCS(X_n, Y_m) = (LCS( X_{n-1}, Y_{m-1}), x_n)$

• comma indicates that element, $x_n$, is appended to the sequence. Note that the LCS for $X_n$ and $Y_m$ involves determining the LCS of the shorter sequences, $X_{n-1}$ and $Y_{m-1}$.
Longest common subsequence

Suppose that the two sequences X and Y do not end in the same symbol.

- LCS of X and Y is the longer of the two sequences LCS(X_n,Y_{m-1}) and LCS(X_{n-1},Y_m).
- Example: sequence X: ABCDEFG (n elements) sequence Y: BCDGK (m elements)
- Last character of the LCS of these two sequences either ends with a G (the last element of sequence X) or does not.

**Case 1: the LCS ends with a G**
Then it cannot end with a K. Thus remove the K from sequence Y: if K were in the LCS, it would be its last character; as a consequence K is not in the LCS:
- LCS(X_n,Y_m) = LCS(X_n, Y_{m-1}).

**Case 2: the LCS does not end with a G**
Then remove the G from the sequence X: LCS(X_n,Y_m) = LCS(X_{n-1}, Y_m).
- In any case, LCS is LCS(X_n, Y_{m-1}) or LCS(X_{n-1}, Y_m).
- Those two last LCS are both common subsequences to X and Y. LCS(X,Y) is the longest.
- Thus longest sequence of LCS(X_n, Y_{m-1}) and LCS(X_{n-1}, Y_m).
LCS function

Let two sequences be defined as follows: \( X = (x_1, x_2...x_m) \) and \( Y = (y_1, y_2...y_n) \). The prefixes of \( X \) are \( X_{1, 2,...m} \); the prefixes of \( Y \) are \( Y_{1, 2,...n} \). Let \( LCS(X_i, Y_j) \) represent the set of longest common subsequence of prefixes \( X_i \) and \( Y_j \). This set of sequences is given by the following.

- empty set if \( i=0 \) or \( j=0 \)
- \( LCS(X_iY_j) = (LCS(X_{i-1}, Y_{j-1}), x_i) \) if \( x_i=y_i \)
  longest \( (LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) \) if \( x_i \neq y_i \)

To find the longest subsequences common to \( X_i \) and \( Y_j \), compare the elements \( x_i \) and \( y_j \).
- If they are equal, then the sequence \( LCS(X_{i-1}, Y_{j-1}) \) is extended by that element, \( x_i \).
- Else
  - then the longer of the two sequences, \( LCS(X_i, Y_{j-1}) \), and \( LCS(X_{i-1}, Y_j) \), is retained.
  - (If they are both the same length, but not identical, then both are retained.)
LCS function

• Q3. Write a programme to solve the LCs problem in $O(nm)$, $n,m$ lengths of the input sequences

• The function below takes as input sequences $X[1..m]$ and $Y[1..n]$.

• computes the LCS between $X[1..i]$ and $Y[1..j]$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$, and stores it in $C[i,j]$.

• $C[m,n]$ will contain the length of the LCS of $X$ and $Y$.

function LCSLength(X, Y, m, n)
    C = array(0..m, 0..n)
    for i := 0..m C[i,0] = 0
    for j := 0..n C[0,j] = 0
    for i := 1..m
        for j := 1..n
            if $X[i] = Y[j]$
                $C[i,j] := C[i-1,j-1] + 1$
            else
                $C[i,j] := \max(C[i,j-1], C[i-1,j])$
    return $C[m,n]$
C. Optimal binary search trees

• Let a set of words $k_1 < \ldots < k_n$ and their respective frequencies: $p_1 \ldots p_n$ (representing the probabilities for respective searches as well).
• Assume we organize them in a binary search tree and we want to search for them with frequencies $p_1 \ldots p_n$ respectively.
• Create a binary search tree that minimizes the cost $p_1d_1 + \ldots + p_nd_n$ where $d_i$ is the depth of the respective node in the tree.
• Some searches may be for values not in K, and so we also have $n+1$ “dummy keys”
• **Q4** Write the pseudo-code of a procedure for calculating the cost of dynamic programming the optimal binary search tree. *Hint: keys in the same subtree are consecutive*
Optimal binary search trees example taken from [1].

Two binary search trees
- \((k_i, p_i)\) = (query for existing key, probability of \(k_i\))
- \((d_i, q_i)\) = (query for non-existing key, probability of \(d_i\))

According to 15.11 the total in search cost for a: 2.80.
for b: 2.75.

This tree is optimal.

Building the optimal search tree

• Assume keys $k_i \ldots k_j$ and dummy keys $d_{i-1} \ldots d_j$.
• one of these keys, say $k_r$ ($i \leq r \leq j$), is the root of an optimal subtree containing these keys.
• left subtree: $\{k_i \ldots k_r\}$ (and dummy keys $d_{i1} \ldots d_{r1}$), right subtree $k_{r+1} \ldots k_j$ (and dummy keys $d_{r} \ldots d_{j}$).
• examine all candidate roots $k_r$, determine all optimal binary search trees containing $k_i \ldots k_{r-1}$ and those containing $k_{r+1} \ldots k_j$.
• find an optimal binary search tree.
A recursive solution

- \(e[i, j]\) : the expected cost of searching an optimal binary search tree containing the keys \([k_i \ldots k_j]\). Ultimately, we wish to compute \(e[1,n]\).
- Assume root \(k_r\) (\(i \leq r \leq j\)) and left subtree : \(\{k_i \ldots k_{r-1}\}\) right subtree \(k_{r+1} \ldots k_j\)
- The total weight of these nodes are:

\[
    w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l.
\]
- and the total search cost is:

\[
    e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j).
\]
Optimal binary search tree

```
OPTIMAL-BST(p, q, n)
1   let e[1..n+1, 0..n], w[1..n+1, 0..n],
   and root[1..n, 1..n] be new tables
2   for i = 1 to n + 1
3       e[i, i - 1] = q_{i-1}
4       w[i, i - 1] = q_{i-1}
5   for l = 1 to n
6       for i = 1 to n - l + 1
7           j = i + l - 1
8           e[i, j] = \infty
9           w[i, j] = w[i, j - 1] + p_j + q_j
10          for r = i to j
11              t = e[i, r - 1] + e[r + 1, j] + w[i, j]
12             if t < e[i, j]
13                e[i, j] = t
14                root[i, j] = r
15   return e and root
```

Root[i,j] stores the root of the optimal search tree for the contiguous nodes [1,...j]

2-4 : initialisation of the cost and weight matrices
5-14 : uses recurrence to compute e[i,j] and w[i,j] for all 1<=i<=j<=n
- for i=1 computes all e[i,i] and w[i,i] for i=1..n
- for i=2 computes all e[i,i+1] and w[i,i+1] for i=1...n-1
- ...

10-14: try to find the best root r (minimizing the cost of the tree containing nodes i..j rooted at r)
Algorithms runs on the example tree