Chapter 6 Dynamic Programming

Yonghui Wu
Shanghai Key Laboratory of Intelligent Information Processing
School of Computer Science, Fudan University
yhwu@fudan.edu.cn

Dynamic Programming

- This lecture is supported by Office of Global Partnerships (Key Projects Development Fund), Fudan University.

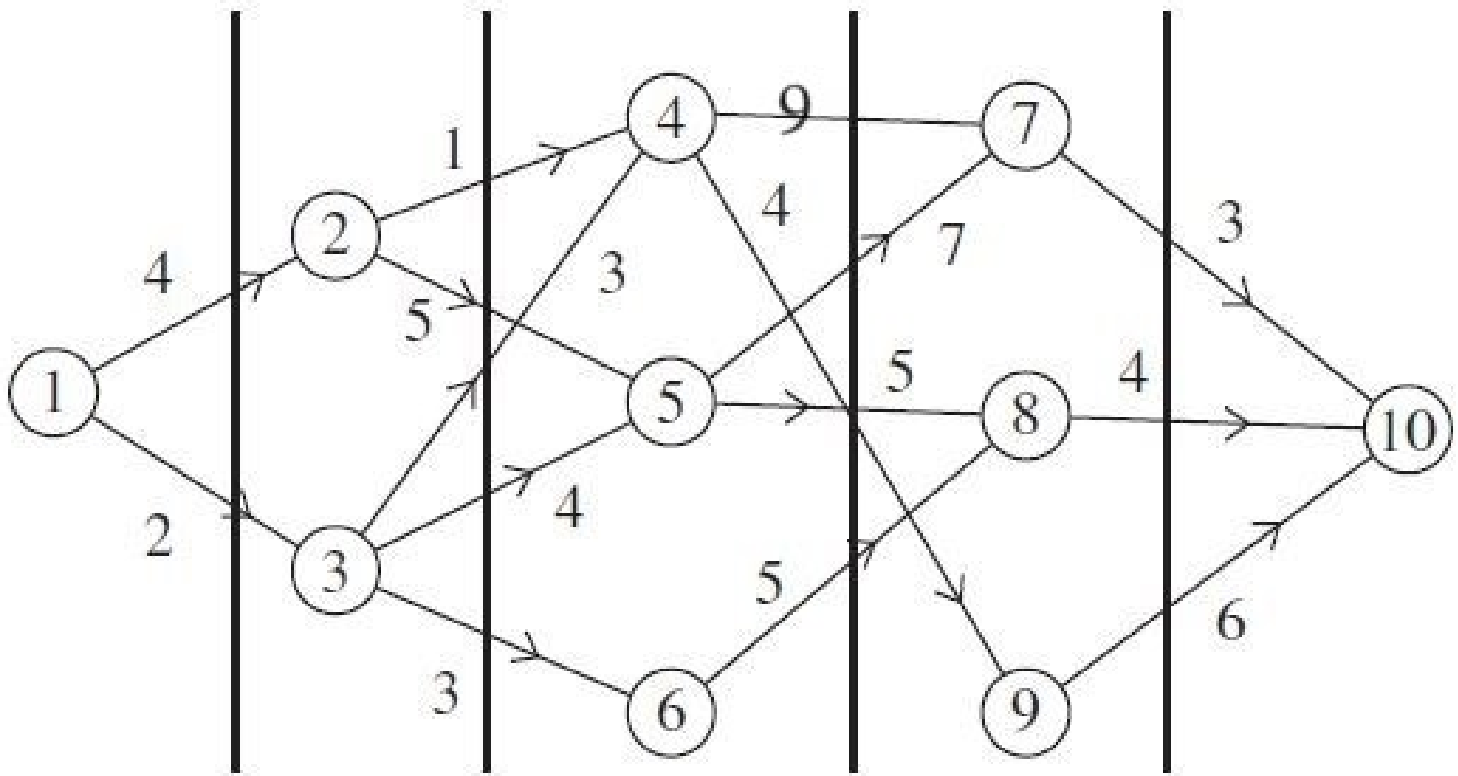
Dynamic Programming

- Dynamic Programming (DP)
  - solve optimization problems.
- Dynamic programming breaks an optimization problem into a sequence of related subproblems, solves these subproblems just once, stores solutions to subproblems, and constructs an optimal solution to the problem based on solutions to subproblems.
- The method storing solutions to subproblems is called memorization. When the same subproblem occurs, its solution can be used directly.

Dynamic Programming

- Two characters for a problem solved by Dynamic Programming.
  - Optimization. An optimal solution to a problem consists of optimal solutions to subproblems.
  - No aftereffect. A solution to a subproblem is only related to solutions to its direct predecessors.

Example : Dijkstra algorithm



Chapter 6 Practice for Dynamic Programming

- 6.1 Linear Dynamic Programming
  - 6.1.1 Linear Dynamic Programming
  - 6.1.2 Subset Sum
  - 6.1.3 Longest Common Subsequence (LCS)
  - 6.1.4 Longest Increasing Subsequence (LIS)
- 6.2 Tree-Like Dynamic Programming
- 6.3 Dynamic Programming with State Compression

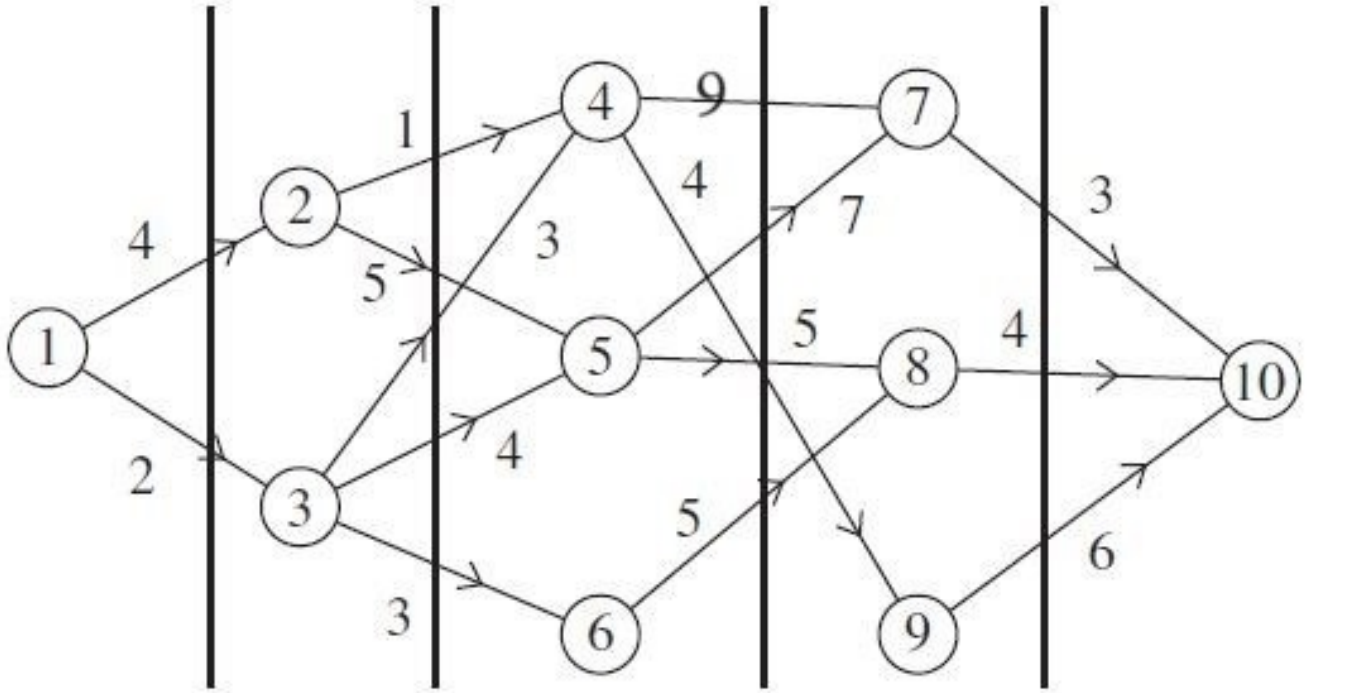
6.1 Linear Dynamic Programming

- 6.1.1 Linear Dynamic Programming
- 6.1.2 Subset Sum
- 6.1.3 Longest Common Subsequence (LCS)
- 6.1.4 Longest Increasing Subsequence (LIS)

6.1.1 Linear Dynamic Programming

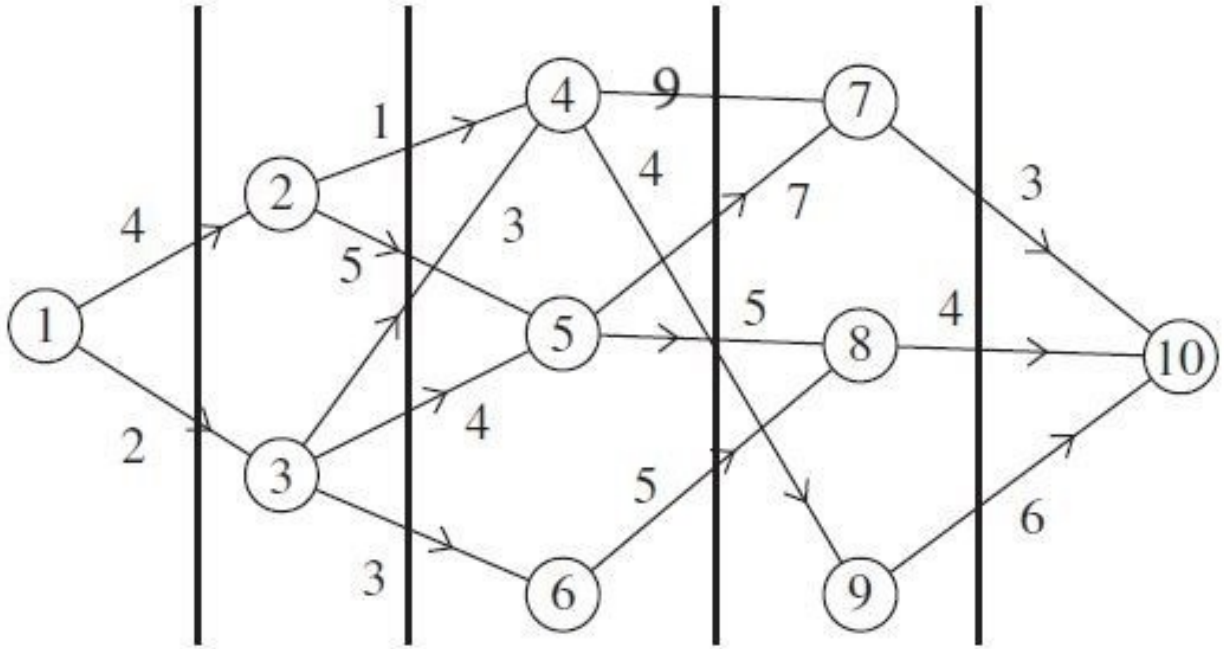
- **Stage k and State s_k:**
  - The solution to a problem is divided into k orderly and related stages.
  - In a stage there are several states.
  - State s_k is a state in stage k.

- A problem is divided into 5 orderly and related stages.
  - State 1 is called the initial state, in stage 1.
  - State 10 is called the goal state, in stage 5.
  - In stage 3 there are 3 states: state 4, state 5, and state 6.



- **Decision u_k and Available Decision Set $D_k(s_k)$:**
  - The choice from a state in stage k-1 (the current stage) to a state in stage k (the next stage) is called decision u_k.
  - A state can be reachable through more than one decision from the last stage, and such decisions constitute an available decision set $D_k(s_k)$.

- Two decisions reaching state 5: $2 \rightarrow 5$, $3 \rightarrow 5$, $D_3(5)=\{2, 3\}$.
- A decision sequence from the initial state to the goal state is called a strategy.
  - $1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 10$ is a strategy.



- **Successor Function and Optimization:**
  - A successor function :
    - describe the transition from stage *k*-1 to stage *k*.
- Dynamic programming method is used to solve some optimization problems.
  - Successor functions :
    - find a solution with the optimal (minimum or maximum) value to a problem.

- A successor function

$$f_k(s_k) = \quad opt \quad g(f_{k-1}(T_k(s_k, u_k)), u_k)$$

- where
  - $T_k(s_k, u_k)$ is a state s_{k-1} in stage k-1 which relate to state s_k through decision u_k,
  - $f_{k-1}(T_k(s_k,u_k))$ is an optimal solution,
  - $g(x, u_k)$ is a function for value x and decision u_k, that is, $g(f_{k-1}(T_k(s_k, u_k)$ is a function from state s_{k-1} to state s_k through decision u_k;
  - *opt* means optimization;
  - $f_1(s_1)$ is an initial value.
  - u_k is one decision in decision set $D_k(s_k)$, all decisions are enumerated to get the optimal solution to s_k.
- From initial state, successor functions are used to get the optimal solution f_n (goal state) to the problem finally.

linear DP

for (every stage i is processed in linear order)

for （ every state j in stage i is enumerated（$j \in S_i$））

for（ every state k in stage i-1 which is related to state j is enumerated ($k \in S_{i\text{-}1}$)）

calculate $f_i(j) = \underset{u_k \in D_k(k)}{opt}\ g\left(f_{i-1}\left(k\right), u_k\right);$

6.1.1.1 Brackets Sequence

- Source: ACM Northeastern Europe 2001
- IDs for Online Judges: POJ 1141， ZOJ 1463， Ural 1183， UVA 2451

- Let us define a regular brackets sequence in the following way:
- 1. Empty sequence is a regular sequence.
- 2. If S is a regular sequence, then (S) and $[S]$ are both regular sequences.
- 3. If A and B are regular sequences, then AB is a regular sequence.
- For example, all of the following sequences of characters are regular brackets sequences:
- (), [], (()), ([]), ()[], ()[()]
- And all of the following character sequences are not:
- (, [,),)(, ([)], ([(]
- Some sequence of characters '(', ')', '[', and ']' is given. You are to find the shortest possible regular brackets sequence, that contains the given character sequence as a subsequence. Here, a string $a_1\ a_2\ \ldots\ a_n$ is called a subsequence of the string $b_1\ b_2\ \ldots\ b_m$, if there exist such indices $1 \le i_1 < i_2 < \ldots < i_n \le m$, that $a_j = b_{ij}$ for all $1 \le j \le n$.

- **Input**
- The input file contains at most 100 brackets (characters '(', ')', '[' and ']') that are situated on a single line without any other characters among them.
- **Output**
- Write to the output file a single line that contains some regular brackets sequence that has the minimal possible length and contains the given sequence as a subsequence.

Analysis

Suppose stage r is the length of subsequence, $1 \le r \le n$; and state i is the pointer pointing to the front of the current subsequence, $0 \le i \le n-r$. Based on i and r, the pointer j pointing to the rear of the current subsequence can be calculated, $j=i+r-1$. Suppose $dp[i, j]$ is the minimal number of characters that must be inserted into $s_i \ldots s_j$. Obviously, if the length of subsequence is 1, $dp[i, i]=1$, $0 \le i <$ strlen(s).

If ((s_i=='[') && (s_j==']') || (s_i=='(')&& (s_j==')')), then the minimal number of characters that must be inserted into $s_i \ldots s_j$ is the minimal number of characters that must be inserted into $s_{i+1} \ldots s_{j-1}$, that is, $dp[i, j]=dp[i+1, j-1]$; otherwise $s_i \ldots s_j$ is divided into two parts, and we need to determine the pointer k ($i \le k < j$) so that $dp[i,j]= \min_{1 \le k < j} (dp[i, k]+dp[k+1, j])$.

Based on it, a memorized list *path*[ ][ ] is used to store all solutions to subproblems:

$$path[i][j]=\begin{cases} -1 & ((s_i == '[') \&\& (s_j == ']')) \| ((s_i == '(') \&\& (s_j == ')')) \\ k & dp[i,j] = \min_{i \le k < j}(dp[i,k] + dp[k+1,j]) \end{cases}$$

After memorized list *path*[ ][ ] is calculated through DP, regular brackets sequence that has the minimal possible length and contains the given sequence as a subsequence can be gotten through recursion.

6.1.2 Subset Sum

- Suppose $S=\{x_1, x_2, \ldots, x_n\}$ is a set of non-negative integers, and c is a non-negative integer.
- Subset Sum Problem
  - determine whether there is a subset of the given set with sum equal to given c.

Coin Counting

- Given a set of n non-negative integers $\{a_1, a_2, ..., a_n\}$ and a non-negative integer T,
- Coin Counting
  - determine how many solutions to $k_1a_1+k_2a_2+\ldots+k_na_n=T$, where k_1 , k_2 , ..., k_n are non-negative integers.

DP used to solve Coin Counting

- Suppose $c(i, j)$ is the number of solutions to $k_1a_1+k_2a_2+\ldots+k_ia_i=j$, $k_i>0$.
- The goal for Coin Counting is to calculate $c(n, T)$.
- Calculate $c(i, j)$,
  - stage i is the first i integers are used, $1\leq i\leq n$;
  - states are $k_1a_1+k_2a_2+\ldots+k_ia_i=j$, $a_i\leq j\leq T$.
- The successor function
  - $c[i][j]=c[i-1][j]+c[i][j-a_i]$

6.1.2.1 Dollars

- Source: New Zealand Contest 1991
- IDs for Online Judge: UVA 147

- New Zealand currency consists of $100, $50, $20, $10, and $5 notes and $2, $1, 50c, 20c, 10c and 5c coins. Write a program that will determine, for any given amount, in how many ways that amount may be made up. Changing the order of listing does not increase the count. Thus 20c may be made up in 4 ways: 1×20c, 2×10c, 10c+2×5c, and 4×5c.

- **Input**
- Input will consist of a series of real numbers no greater than $50.00 each on a separate line. Each amount will be valid, that is will be a multiple of 5c. The file will be terminated by a line containing zero (0.00).
- **Output**
- Output will consist of a line for each of the amounts in the input, each line consisting of the amount of money (with two decimal places and right justified in a field of width 5), followed by the number of ways in which that amount may be made up, right justified in a field of width 12.

Analysis

- First, DP is used to calculate all solutions to the problem in the range.
  - 5c coin is the smallest coin. Other notes and coins for New Zealand currency are multiples for 5c coin.
  - 5c coin is used as the unit for notes and coins for New Zealand currency.
  - $b[i]$ is the number of 5c coins for the i-th currency, $1 \le i \le 11$;
  - $a[j]$ is the number of ways in which j 5c coins may be made up using the first i-th currencies. Initially, $a[j] = 1$; $0 \le j \le 6000$.

- $a[j]$ (the number of ways in which j 5c coins may be made up using the first i-th currencies)
  - $a[j]$ (the number of ways in which j 5c coins may be made up using the first (i-1)-th currencies)
  - $a[j-b[i]]$ (the number of ways in which j 5c coins may be made up using the first (i-1)-th currencies and at least one i-th currency)
- $a[j]+=a[j-b[i]]$。

- Every test case is dealt with based on array a:
  - Each amount n ,
  - The solution $a[\lfloor n \times 20 \rfloor]$

6.1.3 Longest Common Subsequence (LCS)

- For a sequence, elements in its subsequence appear in the same relative order, and aren't necessarily contiguous.
  - For the string "abcdefg", "abc", "abg", "bdf", and "aeg" are all subsequences.
  - For strings "HIEROGLYPHOLOGY" and "MICHAELANGELO", string "HELLO" is a common subsequence.
- Given two sequence of items, Longest Common Subsequence (LCS) is to find the longest subsequence in both of them.

- LCS problem can be solved in terms of smaller subproblems.

- Given two sequences x and y, of length m and n respectively.
- Sequence $x=<x_1, x_2, .., x_m>$,
  - the *i*-th prefix $x_i'==<x_1, x_2, .., x_i>$, $i=0,1,..,m$;
- Sequence $y=<y_1, y_2, .., y_n>$,
  - the *i*-th prefix $y_i'==<y_1, y_2, .., y_i>$, $i=0,1,..,n$;
- sequence $z=<z_1, z_2, .., z_k>$ is a LCS for x and y.

- if $x=<A,B,C,B,D,A,B>$, then $x_4'=<A,B,C,B>$, and x_0' is an empty sequence.

- Stage and state are pointer i for prefix of x and pointer j for prefix of y respectively.
- x_{i-1} and y_{j-1} have been calculated through LCS.
- Decisions are made based on following properties.
  - **Property 1.** If $x_m = y_n$, then $z_K = x_m = y_n$ and z_{k-1}' is a LCS for x_{m-1}' and y_{n-1}'.
  - **Property 2.** If $x_m \neq y_n$, then $z_K \neq x_m$, and z is a LCS for x_{m-1}' and y.
  - **Property 3.** If $x_m \neq y_n$, then $z_K \neq y_n$, and z is a LCS for x and y_{n-1}'.

- Suppose $c[i, j]$ is the length of LCS for x_i' and y_j'.

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

6.1.3.1 Longest Match

- **Source: TCL Programming Contest 2001**
- **IDs for Online Judge: UVA 10100**

- A newly opened detective agency is struggling with their limited intelligence to find out a secret information passing technique among its detectives. Since they are new in this profession, they know well that their messages will easily be trapped and hence modified by other groups. They want to guess the intensions of other groups by checking the changed sections of messages. First they have to get the length of longest match. You are going to help them.

- **Input**
- The input file may contain multiple test cases. Each case will contain two successive lines of string. Blank lines and non-letter printable punctuation characters may appear. Each Line of string will be no longer than 1000 characters. Length of each word will be less than 20 characters.
- **Output**
- For each case of input, you have to output a line starting with the case number right justified in a field width of two, followed by the longest match as shown in the sample output. In case of at least one blank line for each input output 'Blank!'. Consider the non-letter punctuation characters as white-spaces.

Analysis

- Consecutive letters in a string is regarded as a word.
- Words in two strings are gotten one by one, where words in the first string are stored in *T*1.*word*[1]… *T*1.*word*[*n*], and words in the second string are stored in *T*2.*word*[1]…*T*2.*word*[*m*].

- Every word is regarded as a “character”.
- LCS algorithm is used to calculate the Longest Common Subsequence (LCS).
- The length of the subsequence is the length of longest match.

6.1.4 Longest Increasing Subsequence (LIS)

- The Longest Increasing Subsequence (LIS) problem is to find the longest increasing subsequence of a given sequence.
- Given a real sequence $A=<a_1, a_2, \ldots, a_n>$, the Longest Increasing Subsequence for A is such a longest subsequence $L=<a_{k1}, a_{k2}, \ldots, a_{km}>$, where $k_1<k_2<\ldots<k_m$ and $a_{k1}<a_{k2}<\ldots<a_{km}$.

- **A LIS problem is transformed into a LCS problem.**
  - Suppose $X=<b_1, b_2, \ldots, b_n>$ is a sorted sequence in ascending order for $A=<a_1, a_2, \ldots, a_n>$.
  - The LCS for X and A is the LIS for A.

- The time complexity for sorting A is $O(n\log_2(n))$. The time complexity calculating the LCS for X and A is $O(n^2)$. Therefor the time complexity for the **Method** is $O(n\log_2(n)+n^2)$.

6.1.4.1 History Grading

- Source: Internet Programming Contest 1991
- IDs for Online Judge: UVA 111

- Many problems in Computer Science involve maximizing some measure according to constraints. Consider a history exam in which students are asked to put several historical events into chronological order. Students who order all the events correctly will receive full credit, but how should partial credit be awarded to students who incorrectly rank one or more of the historical events?

- Some possibilities for partial credit include:
  - 1 point for each event whose rank matches its correct rank;
  - 1 point for each event in the longest (not necessarily contiguous) sequence of events which are in the correct order relative to each other.
- For example, if four events are correctly ordered 1 2 3 4 then the order 1 3 2 4 would receive a score of 2 using the first method (events 1 and 4 are correctly ranked) and a score of 3 using the second method (event sequences 1 2 4 and 1 3 4 are both in the correct order relative to each other).

- In this problem you are asked to write a program to score such questions using the second method.
- Given the correct chronological order of n events 1, 2, …, n as $c_1, c_2, \ldots, c_n$ where $1 \le c_i \le n$ denotes the ranking of event i in the correct chronological order and a sequence of student responses $r_1, r_2, \ldots, r_n$ where $1 \le r_i \le n$ denotes the chronological rank given by the student to event i; determine the length of the longest (not necessarily contiguous) sequence of events in the student responses that are in the correct chronological order relative to each other.

- **Input**
- The first line of the input will consist of one integer n indicating the number of events with $2 \leq n \leq 20$. The second line will contain n integers, indicating the correct chronological order of n events. The remaining lines will each consist of n integers with each line representing a student's chronological ordering of the n events. All lines will contain n numbers in the range [1..n], with each number appearing exactly once per line, and with each number separated from other numbers on the same line by one or more spaces.
- **Output**
- For each student ranking of events your program should print the score for that ranking. There should be one line of output for each student ranking.

Analysis

- Suppose *st*[ ] is the correct chronological order of *n* events, where *st*[*t*] is the *t*-th event in the chronological order; *ed*[ ] is the current student's chronological ordering of the *n* events, where *ed*[*t*] is the *t*-th event in the current student's chronological order.
- The Longest Common Subsequence (LCS) for *st*[ ] and *ed*[ ] is the Longest Increasing Subsequence (LIS) for *ed*[ ], where its length is the score for that ranking. **Method 1** is used to solve the problem.

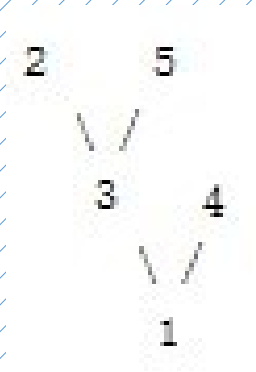
6.2 Tree-Like Dynamic Programming

- If the background or the relationships between stages for a DP problem are represented as a tree, tree-like dynamic programming can be used to solve such problems.
- Tree-like dynamic programming is different with linear DP:
  - The calculation sequences are different. There are two calculation sequences for linear DP: Forward and Backward. The calculation sequence for tree-like dynamic programming is normally from leafs to the root, and the root is the solution.
  - The calculation methods are different. Traditional iteration method is used in Linear DP. Recursive method is used in tree-like dynamic programming, for tree-like dynamic programming is normally implemented by memorized search.

6.2.1 Binary Apple Tree

- **Source: Ural State University Internal Contest '99 #2**
- **IDs for Online Judge: Ural 1018**

- Let's imagine how apple tree looks in binary computer world. You're right, it looks just like a binary tree, i.e. any biparous branch splits up to exactly two new branches. We will enumerate by integers the root of binary apple tree, points of branching and the ends of twigs. This way we may distinguish different branches by their ending points. We will assume that root of tree always is numbered by 1 and all numbers used for enumerating are numbered in range from 1 to *N*, where *N* is the total number of all enumerated points. For instance in the picture below *N* is equal to 5. Here is an example of an enumerated tree with four branches:



- As you may know it's not convenient to pick an apple from a tree when there are too much of branches. That's why some of them should be removed from a tree. But you are interested in removing branches in the way of minimal loss of apples. So you are given amounts of apples on a branches and amount of branches that should be preserved. Your task is to determine how many apples can remain on a tree after removing of excessive branches.

- **Input**
- First line of input contains two numbers: N and Q ($2 \le N \le 100$; $1 \le Q \le N - 1$). N denotes the number of enumerated points in a tree. Q denotes amount of branches that should be preserved. Next $N - 1$ lines contains descriptions of branches. Each description consists of a three integer numbers divided by spaces. The first two of them define branch by it's ending points. The third number defines the number of apples on this branch. You may assume that no branch contains more than 30000 apples.
- **Output**
- Output should contain the only number — amount of apples that can be preserved. And don't forget to preserve tree's root ;-)

Analysis

- In this problem, the apple tree is a weighted binary tree. The problem requires you to get such a subtree with Q branches (that is, Q+1 points) whose weight is maximal. For each internal point there are three choices: pruning its left subtree, pruning its right subtree, or pruning some points in its left subtree and its right subtree; to get a subtree with maximal weight.
- Suppose $g[x][k]$ is the maximal weight for the subtree with root x in which there are k points (including the weight of the branch from root x to its parent). For each leaf, DP is used in the order of post-order traversal. The successor function for DP is as follow.

If x is a leaf, then $g[x][k]$= the weight of the edge from x to its parent; else all cases that k-1 nodes are distributed in its left subtree and its right subtree are enumerated, and the best case is found. That is,

$$g[x][k]=\begin{cases}0 & k=0 \\ \text{the weight of the edge from } x \text{ to its parent} & x \text{ is a leaf} \\ \text{the weight of the edge from } x \text{ to its parent} + \max\limits_{0\le i\le k-1}\{g[\text{the left child for } x][i]+g[\text{the right child for } x][k-i-1]\} & x \text{ isn't a leaf}\end{cases}$$

DP is used bottom-up until the root. Finally *ans*=g[*root*][*Q*+1].

6.3 Dynamic Programming with State Compression

- In some problems, each constituent part for a state can be represented as 0 or 1, and states can be represented as strings for 0 and 1. For example, grids in a chessboard can be represented a string. And states for a chessboard can be represented as strings. We call it state compression. Dynamic programming with state compression can be implemented by bitwise operations.

6.3.1 Nuts for nuts..

- **Source: UVA Local Qualification Contest， 2005**
- **IDs for Online Judge: UVA 10944**

- So as Ryan and Larry decided that they don't really taste so good, they realized that there are some nuts located in certain places of the island.. and they love them! Since they're lazy, but greedy, they want to know the shortest tour that they can use to gather every single nut!
- Can you help them?

- **Input**
- You'll be given *x*, and *y*, both less than 20, followed by *x* lines of *y* characters each as a map of the area, consisting sorely of ".", "#", and "L". Larry and Ryan are currently located in "L", and the nuts are represented by "#". They can travel in all 8 adjacent direction in one step. See below for an example. There will be at most 15 places where there are nuts, and "L" will only appear once.
- **Output**
- On each line, output the minimum amount of steps starting from "L", gather all the nuts, and back to "L".

Analysis

Places where Ryan and Larry and all nuts locate are as vertices. Their coordinates are recorded, where (x_0, y_0) is the place where Larry and Ryan are currently located, and (x_i, y_i) is the place where the i-th nut is located, $1 \le i \le n$. $Map[\,][\,]$ is used to represent distances between vertices, where $Map[i][j]=\max\{|x_i - x_j|, |y_i - y_j|\}$ for vertex i and vertex j.

The state that nuts are gathered is represented as a n bit binary number $(b_{n-1},\cdots,b_0)$, where $b_i=0$ means the $(i+1)$-th nut isn't gathered, and $b_i=1$ means the $(i+1)$-th nut is gathered. Suppose j is the current state value that nuts are gathered, where nut i is the nut is gathered finally, and the minimum amount of steps is $f[i][j]$. Obviously the minimum amount of steps that Ryan and Larry gather every nut is $f[i][2^{i-1}]=map[0][i](1 \le i \le n)$. Suppose the state the nuts are gathered currently is j, where the number of gathered nut is i, and the minimum amount of steps is $f[i][j]$. Obviously, the minimum amount of steps that Ryan and Larry gather each nut is $f[i][2^{i-1}]=map[0][i]$, $1 \le i \le n$. The successor function is analyzed as follows.

Stage i; in ascending order states are enumerated, $0 \le i \le 2^n-1$;

State j; The last gathered nut j in stage i is enumerated, $1 \le j \le n$, $i \ \& \ 2^{j-1} \ne 0$;

Decision k: Nut k which isn't in stage i is enumerated ($1 \le k \le n$, $i \ \& \ 2^{k-1} == 0$),and determine whether gathering nut k is better. If gathering nut k is better, $f[k][i+2^{k-1}]$ is adjusted, that is,

$$f[k][i+2^{k-1}] = \min\{ f[k][i+2^{k-1}], f[j][i]+map[j][k] \}.$$

After n nuts are gathered, if nut i is the last gathered nut, the minimum amount of steps that reaching the position of nut i is $f[i][2^n-1]$, the number of steps back to "L" is $map[0][i]$. The number of steps is $f[i][2^n-1]+map[0][i]$.

Finally, all results are compared $i(1 \le i \le n)$, the minimum amount of steps starting from "L", gather all the nuts, and back to "L" is

$$ans = \min_{1 \le i \le n} \{ f[i][2^n-1] + map[0][i] \}.$$