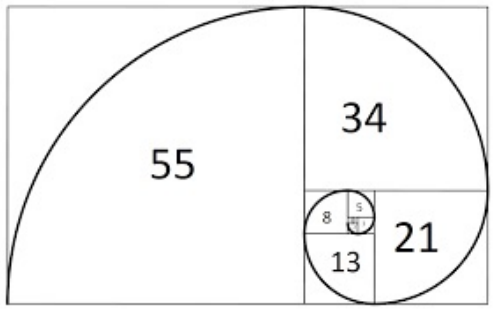
Dynamic programming.

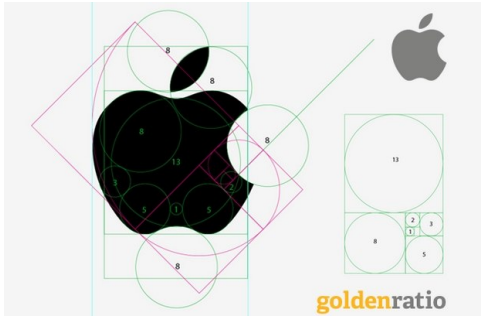
Curs 2018

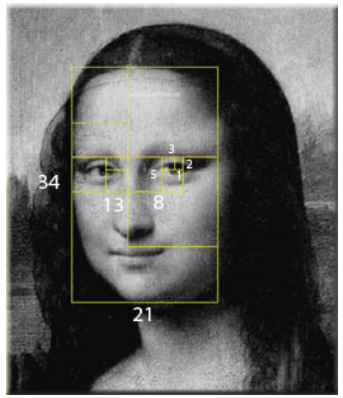
Fibonacci Recurrence.

n-th Fibonacci Term
INPUT: $n \in nat$
QUESTION: Compute
$F_n = F_{n-1} + F_{n-2}$

Recursive Fibonacci (n)
if $n = 0$ **then**
  return 0
else if $n = 1$ **then**
  return 1
else
  (**Fibonacci**$(n-1)+$
  $+$**Fibonacci**$(n-2)$)
end if

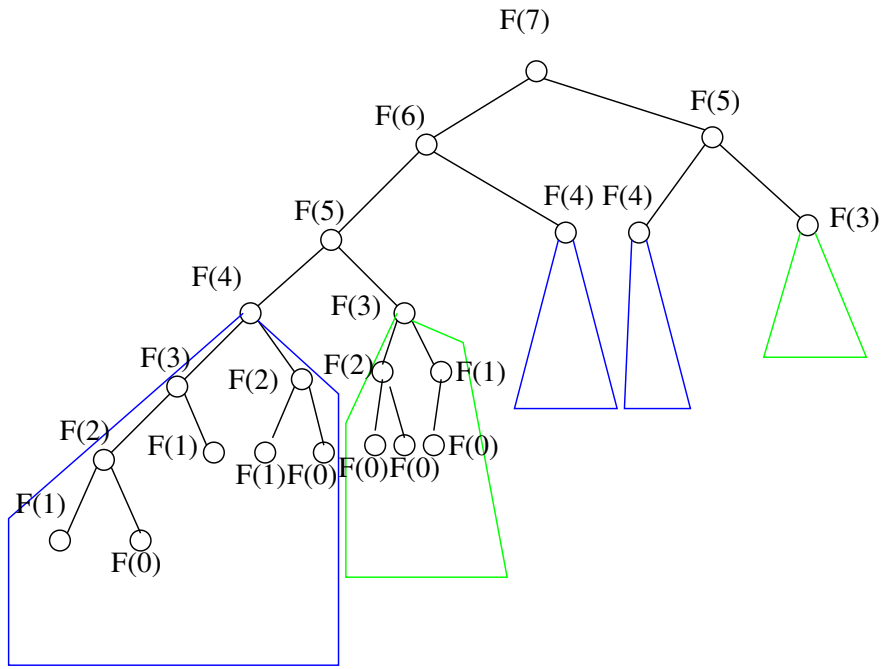






Computing F_7.

As $F_{n+1}/F_n \sim (1+\sqrt{5})/2 \sim 1.61803$ then $F_n > 1.6^n$, and to compute F_n we need 1.6^n recursive calls.



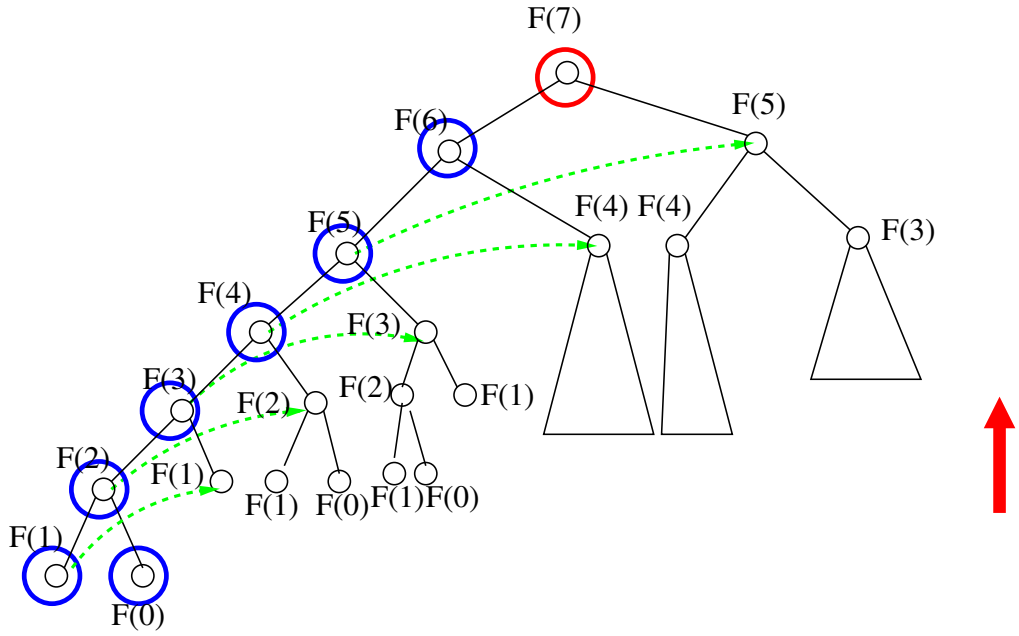
A DP algorithm.

To avoid repeating multiple computations of subproblems, carry the computation bottom-up and store the partial results in a table

DP-Fibonacci (n) {Construct table}
$F_0 = 0$
$F_1 = 1$
for $i = 1$ to n **do**
  $F_i = F_{i-1} + F_{i-2}$
end for

F[0]	0
F[1]	1
F[2]	1
F[3]	2
F[4]	3
F[5]	5
F[6]	8
F[7]	13

To get F_n need $O(n)$ time and $O(n)$ space.



- Recursive (top-down) approach very slow
- Too many identical sub-problems and lots of repeated work.
- Therefore, bottom-up + storing in a table.
- This allows us to look up for solutions of sub-problems, instead of recomputing. Which is more efficient.

Dynamic Programming.

Richard Bellman: *An introduction to the theory of dynamic programming* RAND, 1953
Today it would be denoted Dynamic Planning

Dynamic programming is a powerful technique for efficiently computing *recurrences* by storing partial results and re-using them when needed.

Explore the space of all possible solutions by decomposing things into subproblems, and then building up correct solutions to larger problems.

Therefore, the number of subproblems can be exponential, but if the number of *different problems is polynomial*, we can get a polynomial solution by avoiding to repeat the computation of the same subproblem.

Properties of Dynamic Programming

Dynamic Programming works when:

- Optimal sub-structure: An optimal solution to a problem contains optimal solutions to subproblems.
- Overlapping subproblems: A recursive solution contains a small number of distinct subproblems, repeated many times.

Difference with greedy

- Greedy problems have the greedy choice property: locally optimal choices lead to globally optimal solution.
- For some DP problems greedy choice is not possible globally optimal solution requires back-tracking through many choices.
- I.e. In DP we generate all possible feasible solutions, while in greedy we are bound for the initial choice

Guideline to implement Dynamic Programming

1. *Characterize the structure of an optimal solution:* make sure space of subproblems is not exponential. Define variables.
2. Define recursively the value of an optimal solution: Find the correct recurrence, with solution to larger problem as a function of solutions of sub-problems.
3. *Compute, bottom-up, the cost of a solution:* using the recursive formula, tabulate solutions to smaller problems, until arriving to the value for the whole problem.
4. *Construct an optimal solution:* Trace-back from optimal value.

Implementtion of Dynamic Programming

Memoization: technique consisting in storing the results of subproblems and returning the result when the same sub-problem occur again. Technique used to speed up computer programs.

- In implementing the DP recurrence using recursion could be very inefficient because solves many times the same sub-problems.
- But if we could manage to solve and store the solution to sub-problems without repeating the computation, that could be a clever way to use recursion + memoization.
- To implement memoization use any dictionary data structure, usually tables or hashing.

Implementation of Dynamic Programming

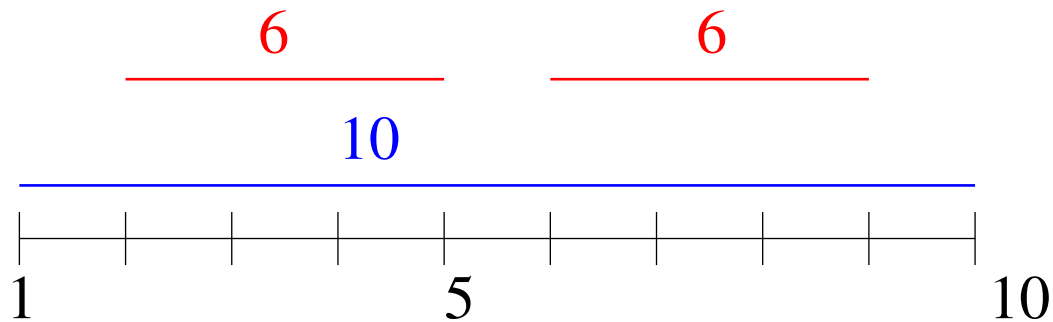
- The other way to implement DP is using iterative algorithms.
- DP is a trade-off between time speed vs. storage space.
- In general, although recursive algorithms ha exactly the same running time than the iterative version, the constant factor in the O is quite more larger because the overhead of recursion. On the other hand, in general the memoization version is easier to program, more concise and more elegant.

Top-down: Recursive and Bottom-up: Iterative

Weighted Activity Selection Problem

Weighted Activity Selection Problem
INPUT: a set $S = \{1, 2, \ldots, n\}$ of activities to be processed by a single resource. Each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$, and a weight w_i.
QUESTION: Find the set of mutually compatible such that it maximizes $\sum_{i \in S} w_i$

Recall: Greedy strategy not always solved this problem.

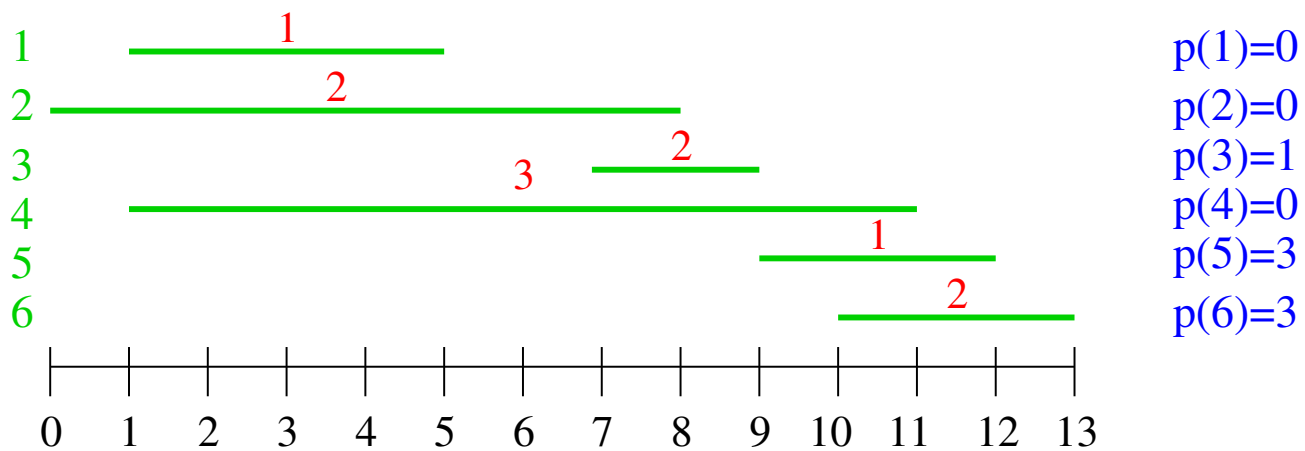


Notation for the weighted activity selection problem

We have $\{1, 2, \ldots, n\}$ activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights $\{w_i\}$.
Therefore, we may need a $O(n \lg n)$ pre-processing sorting step.

Define $p(j)$ to be the largest integer $i < j$ such that i and j are disjoints ($p(j) = 0$ if no disjoint $i < j$ exists).

Let Opt(j) be the value of the optimal solution to the problem consisting of activities in the range 1 to j. Let O_j be the set of jobs in optimal solution for $\{1, \ldots, j\}$.



Recurrence

Consider sub-problem $\{1, \ldots, j\}$. We have two cases:

1.- $j \in O_j$:

- w_j is part of the solution,
- no jobs $\{p(j)+1, \ldots, j-1\}$ are in O_j,
- if $O_{p(n)}$ is the optimal solution for $\{1, \ldots, p(n)\}$ then $O_j = O_{p(n)} \cup \{j\}$ (optimal substructure)

2.- $j \notin O_j$: then $O_j = O_{j-1}$

$$\text{Opt}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{(\text{Opt}(p(j)) + w_j), \text{Opt}(j-1)\} & \text{if } j \geq 1 \end{cases}$$

Recursive algorithm

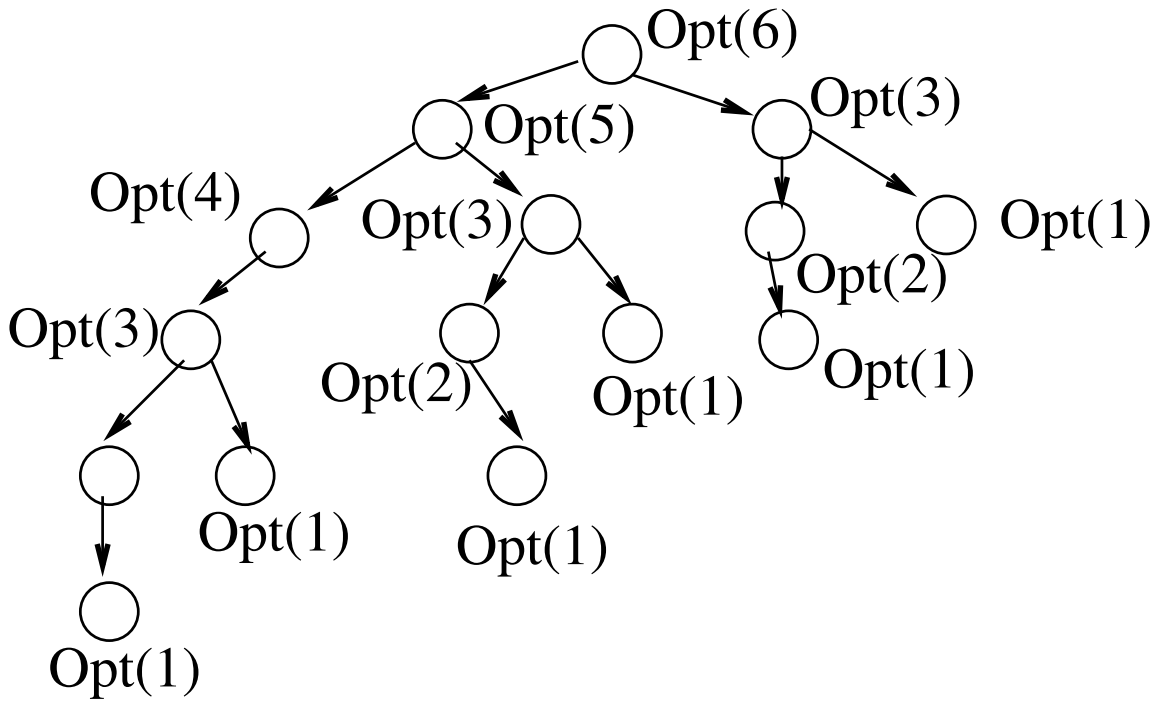
Considering the set of activities S, we start by a pre-processing phase: sorting the activities by increasing $\{f_i\}_{i=1}^{n}$ and computing and tabulating $P[1, \ldots, n] = \{p[j]\}$.
The cost of the pre-computing phase is: $O(n \lg n + n)$

Therefore we assume S is sorted and all $p(j)$ are computed and tabulated in $P[1 \cdots n]$

To compute Opt(j):

R-Opt (j)
if $j = 0$ **then**
  return 0
else
  return $\max(w_j + \textbf{R-Opt}(p(j)), \textbf{R-Opt}(j-1))$
end if

Recursive algorithm



What is the worst running time of this algorithm?: $O(2^n)$

Iterative algorithm

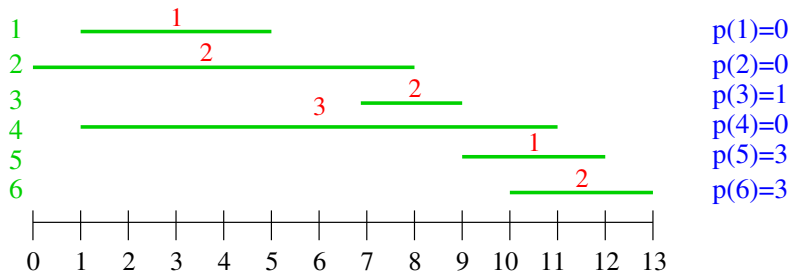
Assuming we have as input the set S of n activities sorted by increasing f, each i with s_i, w_i and the values of $p(j)$, we define through the process a $1 \times (n+1)$ table $M[0, 1 \ldots, n]$, where $M[i]$ contains the value to the best partial solution from 1 to i.

Opt-Val (n)
Define table $M[]$
$M[0] = 0$
for $j = 1$ to n **do**
  $M[j] = \max(M[P[j]] + w_j, M[j-1])$
end for
return $M[n]$

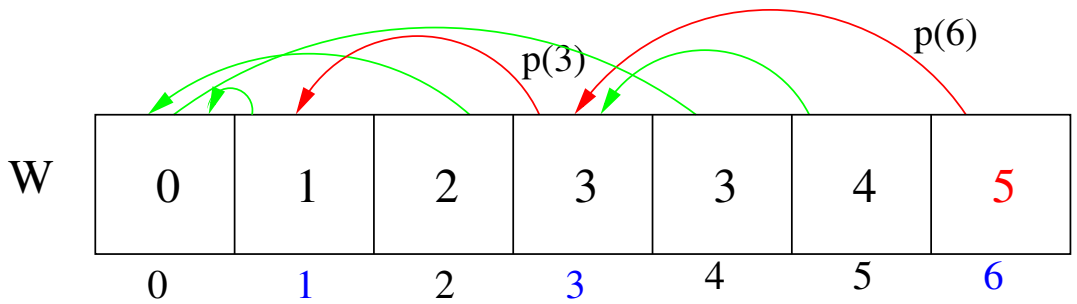
Notice: this algorithm gives only the numerical max. weight

Time complexity: $O(n)$ (not counting the pre-process)

Iterative algorithm: Example



i	s_i	f_i	w_i	P
1	1	5	1	0
2	0	8	2	0
3	7	9	2	1
4	1	11	3	0
5	9	12	1	3
6	10	13	2	3



Sol.: Max weight=5

The DP algorithm

To get also the list of selected activities:

Find-Opt (j)
if $j = 0$ **then**
  return 0
else if $M[P[j]] + w_j > M[j-1]$ **then**
  return j together with Find-Opt$(P[j])$
else
  return Find-Opt$(j-1)$
end if

Time complexity: $O(n)$

0-1 Knapsack

0-1 Knapsack
INPUT:a set $I = \{i\}_1^n$ of items that can NOT be fractioned, each i with weight w_i and value v_i. A maximum weight W permissible
QUESTION: select the items $S \subseteq I$ to maximize the profit.

Recall that we can NOT take fractions of items.

Characterize structure of optimal solution and define recurrence

As part of the input, we need two variables i (the item) and w (the cumulative weight up to item i).
Let v be a variable indicating the optimal value we have obtained so far.

Let us first compute the optimal value $v[n, W]$, and later we compute the set S of objets that yield that value.

Let $v[i, w]$ be the maximum value (optimum) we can get from objects $\{1, 2, \ldots, i\}$ within total weight $\leq w$.

We wish to compute $v[n, W]$.

Recurrence

To compute $v[i, w]$ we have two possibilities:

- That the i-th element is not part of the solution, then we go to compute $v[i-1, w]$,
- or that the i-th element is part of the solution. we add v_i substract w_i to the remaining weight and call $v[i-1, w-w_i]$.

This gives the recurrence,

$$v[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ v[i-1, w-w_i] + v_i & \text{if } i \text{ is part of the solution} \\ v[i-1, w] & \text{otherwise} \end{cases}$$

i.e. $v[i, w] = \max\{v[i-1, w-w_i] + v_i, v[i-1, w]\}$

DP algorithm

Define a table $M = v[0 \ldots n, 0 \ldots W]$,

Knapsack(i, w)
for $i = 1$ **to** $n - 1$ **do**
  $v[i, 0] := 0$
end for
for $i = 1$ **to** n **do**
  for $w = 0$ **to** W **do**
    if $w_i > w$ **then**
      $v[i, w] := v[i - 1, w]$
    else
      $v[i, w] := \max\{v[i - 1, w], v[i - 1, w - w_i] + v_i\}$
    end if
  end for
end for
return $v[n, W]$

The number of steps is $O(nW)$.

Example.

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

$W = 11$.

		w										
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1	1	1	1	1
	2	0	1	6	7	7	7	7	7	7	7	7	7
I	3	0	1	6	7	7	18	19	24	25	25	25	25
	4	0	1	6	7	7	18	22	23	28	29	29	40
	5	0	1	6	7	7	18	22	28	29	34	35	40

For instance,

$v[4,10] = \max\{v[3,10], v[3,11-7]+22\} = \max\{25, 7+22\} = 29.$

$v[5,11] = \max\{v[4,11], v[4,11-7]+22\} = \max\{40, 7+28\} = 40.$

Recovering the solution

To compute the actual subset $S \subseteq I$ that is the solution, we compute with every position $M(i, w)$ a Boolean bit $K(i, w)$, which is 1 iff $i \in S$, otherwise $K(i, w) = 0$.

$X = W$, $S = \emptyset$
for $i = n$ **downto** 1 **do**
  if $K[i, X] = 1$ **then**
    $S = S \cup \{i\}$
    $X = X - w_i$
  end if
end for
Output S

Complexity: $O(nW)$

	0	1	2	3	4	5	6	7	8	9	10	11
0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
1	0 0	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1
2	0 0	1 0	6 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1
3	0 0	1 0	6 0	7 0	7 0	18 1	19 1	24 1	25 1	25 1	25 1	25 1
4	0 0	1 0	6 0	7 0	7 0	18 1	22 1	23 1	28 1	29 1	29 1	40 1
5	0 0	1 0	6 0	7 0	7 0	18 0	22 0	28 1	29 1	34 1	35 1	40 0

$K[5, 11] \to K[4, 11] \to K[3, 5] \to K[2, 0]$. So $S = \{4, 3\}$

Hidden structure in a DP algorithm

Every DP algorithm works because there is an underlying DAG structure, where each node represents a subproblem, and each edge is a precedence constrain on the order in which the subproblems could be solved. Edges could have weights, depending on the problem.

Having nodes $a_1, a_2, \ldots, a_n$ point to b means that b can be solved only when $a_1, a_2, \ldots, a_n$ are solved.

Could you come with the DAG associated to the DP solution of the Knapsack?

Multiplying a Sequence of Matrices

Multiplication of n matrices
INPUT: A sequence of n matrices $(A_1 \times A_2 \times \cdots \times A_n)$
QUESTION: Minimize the number of operation in the computation $A_1 \times A_2 \times \cdots \times A_n$
Recall that Given matrices A_1, A_2 with $\dim(A_1) = p_0 \times p_1$ and $\dim(A_2) = p_1 \times p_2$, the basic algorithm to $A_1 \times A_2$ takes time $p_0 p_1 p_2$ Example:

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

Recall that matrix multiplication is NOT commutative, so we can not permute the order of the matrices without changing the result, but it is associative, so we can put parenthesis as we wish.
In fact, the problem of given $A_1, \ldots, A_n$ with dim $(A_i) = p_{i-1} \times p_i$, how to multiply them to minimize the number of operations is equivalent to the problem of how to parenthesize the sequence $A_1, \ldots A_n$.
Example Consider $A_1 \times A_2 \times A_3$, where dim $(A_1) = 10 \times 100$ dim $(A_2) = 100 \times 5$ and dim $(A_3) = 5 \times 50$.
$((A_1 A_2)A_3) = (10 \times 100 \times 5) + (10 \times 5 \times 50) =$ *7500* operations,
$(A_1(A_2 A_3)) = (100 \times 5 \times 50) + (10 \times 100 \times 50) =$75000 operations.
The order makes a big difference in real computation's time.

How many ways to parenthesize $A_1, \ldots A_n$?

$A_1 \times A_2 \times A_3 \times A_4$:

$(A_1(A_2(A_3A_4))), ((A_1A_2)(A_3A_4)), (((A_1(A_2A_3))A_4),$
$(A_1((A_2A_3)A_4))), (((A_1A_2)A_3)A_4))$

Let $P(n)$ be the number of ways to paranthesize $A_1, \ldots A_n$. Then,

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

with solution $P(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$

The Catalan numbers. Brute force will take too long!

1.- Structure of an optimal solution and recursive solution.

Let $A_{i-j} = (A_i A_{i+1} \cdots A_j)$.
The parenthesization of the subchain $(A_1 \cdots A_k)$ within the optimal parenthesization of $A_1 \cdots A_n$ must be an optimal paranthesization of $A_{k+1} \cdots A_n$.

Notice,
$\forall k, 1 \leq k \leq n, \text{cost}(A_{1-n}) = \text{cost}(A_{1-k}) + \text{cost}(A_{k+1-n}) + p_0 p_k p_n$.

Let $m[i,j]$ the minimum cost of $A_i \times \ldots \times A_j$. Then, $m[i,j]$ will be given by choosing the $k, i \leq k \leq j$ s.t. minimizes
$m[i,k] + m[k+1,j] +$ cost $(A_{1-k} \times A_{k+1-n})$.
That is,

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k \leq j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise} \end{cases}$$

2.- Computing the optimal costs

Straightforward implementation of the previous recurrence:
As $\dim(A_i) = p_{i-1}p_i$, the imput is given by $P =< p_0, p_1, \ldots, p_n >$,

MCR (P, i, j)
if $i = j$ **then**
  return 0
end if
$m[i, j] := \infty$
for $k = i$ **to** $j - 1$ **do**
  $q := \text{MCR}(P, i, k) + \text{MCR}(P, k + 1, j) + p_{i-1}p_k p_j$
  if $q < m[i, j]$ **then**
    $m[i, j] := q$
  end if
end for
return $m[i, j]$.

The time complexity if the previous is given by
$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n \sim \Omega(2^n)$.

Dynamic programming approach.

Use two auxiliary tables: $m[1 \ldots n, 1 \ldots n]$ and $s[1 \ldots n, 1 \ldots n]$.

MCP (P)
for $i = 1$ **to** n **do**
  $m[i, i] := 0$
end for
for $l = 2$ **to** n **do**
  for $i = 1$ **to** $n - l + 1$ **do**
    $j := i + l - 1$
    $m[i, j] := \infty$
    for $k = i$ **to** $j - 1$ **do**
      $q := m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
      if $q < m[i, j]$ **then**
        $m[i, j] := q$, $s[i, j] := k$
      end if
    end for
  end for
end for
return m, s.

$T(n) = \Theta(n^3)$, and space $= \Theta(n^2)$.

Example.

We wish to compute A_1, A_2, A_3, A_4 with $P =< 3, 5, 3, 2, 4 >$
$m[1, 1] = m[2, 2] = m[3, 3] = m[4, 4] = 0$

$i \setminus j$	1	2	3	4
1	0			
2		0		
3			0	
4				0

$l = 2, i = 1, j = 2,$
$k = 1 : q = m[1,2] = m[1,1] + m[2,2] + 3.5.3 = 45\ (A_1A_2)$
$s[i,2] = 1$
$l = 2, i = 2, j = 3,$
$k = 2 : q = m[2,3] = m[2,2] + m[3,3] + 5.3.2 = 30\ (A_2A_3)$
$s[2,3] = 2$
$l = 2, i = 3, j = 4,$
$k = 3 : q = m[3,4] = m[3,3] + m[4,4] + 3.2.4 = 24\ (A_3A_4)$
$s[3,4] = 3$

$i \setminus j$	1	2	3	4
1	0	45		
2	*1*	0	30	
3		*2*	0	24
4			*3*	0

$l = 3, i = 1, j = 3$:

$$m[1,3] = \min \begin{cases} (k=1)m[1,1] + m[2,3] + 3.5.2 = 60 \ A_1(A_2A_3) \\ (k=2)m[1,2] + m[3,3] + 3.3.2 = 63 \ (A_1A_2)A_3 \end{cases}$$

$s[1,3] = 1,\ l = 3, i = 2, j = 4$:

$$m[2,4] = \min \begin{cases} (k=2)m[2,2] + m[3,4] + 5.3.4 = 84 \ A_2(A_3A_4), \\ (k=3)m[2,3] + m[4,4] + 5.2.4 = 70 \ (A_2A_3)A_4. \end{cases}$$

$s[2,4] = 3$

$i \setminus j$	1	2	3	4
1	0	45	60	
2	*1*	0	30	70
3	*1*	*2*	0	24
4		*3*	*3*	0

$l = 4, i = 1, j = 4$:

$$m[1,4] = \min \begin{cases} (k=1) m[1,1] + m[2,4] + 3.5.4 = 130 \ A_1(A_2A_3A_4), \\ (k=2) m[1,2] + m[3,4] + 3.3.4 = 105 \ (A_1A_2)(A_3A_4), \\ (k=3) m[1,3] + m[4,4] + 3.2.4 = 84 \ (A_1A_2A_3)A_4. \end{cases}$$

$i \setminus j$	1	2	3	4
1	0	45	60	84
2	*1*	0	30	70
3	*1*	*2*	0	24
4	*3*	*3*	*3*	0

3.- Constructing an optimal solution

We need to construct an optimal solution from the information in $s[1,\ldots,n,1,\ldots,n]$. In the table, $s[i,j]$ contains k such that the optimal way to multiply:

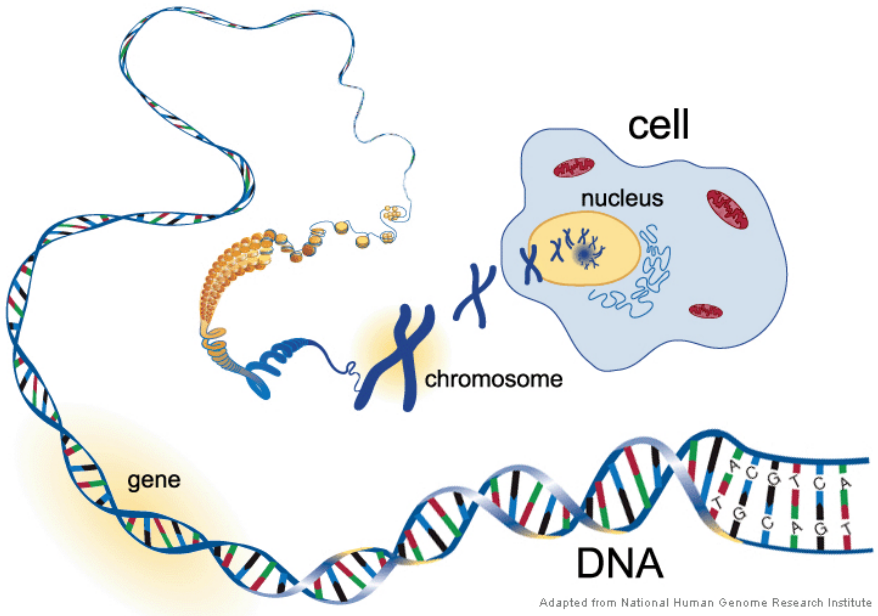
$$A_i \times \cdots \times A_j = (A_i \times \cdots \times A_k)(A_{k+1} \times \cdots \times A_j).$$

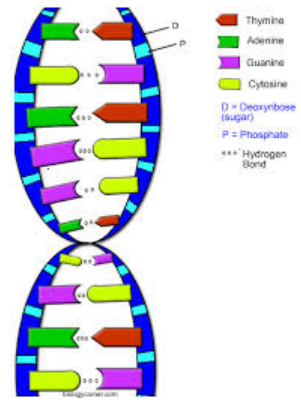
Moreover, $s[i, s[i,j]]$ determines the k to get $A_{i-s[i,j]}$ and $s[s[i,j]+1,j]$ determines the k to get $A_{s[i,j]+1-j}$. Therefore, $A_{1-n} = A_{1-s[1,n]}A_{s[1,n]+1-n}$.

Multiplication(A, s, i, j)
if $j > 1$ **then**
  $X :=$Multiplication $(A, s, i, s[i,j])$
  $Y :=$Multiplication $(A, s, s[i,j]+1, j)$
  return $X \times Y$
else
  return A_i
end if

Therefore $(A_1(A_2A_3))A_4$.

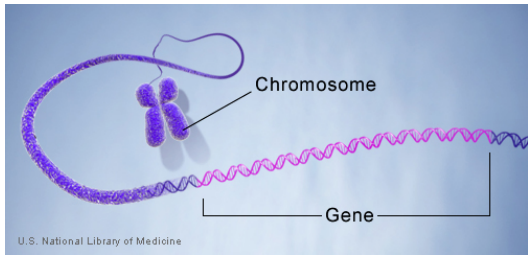
DNA: The book of life





- DNA, is the hereditary material in almost all living organisms. They can reproduce by themselves.
- Its function is like a program unique to each individual organism that rules the working and evolution of the organism.
- El DNA is a string of 3×10^9 characters over $\{A, T, G, C\}^*$.

DNA: Chromosomes and genes

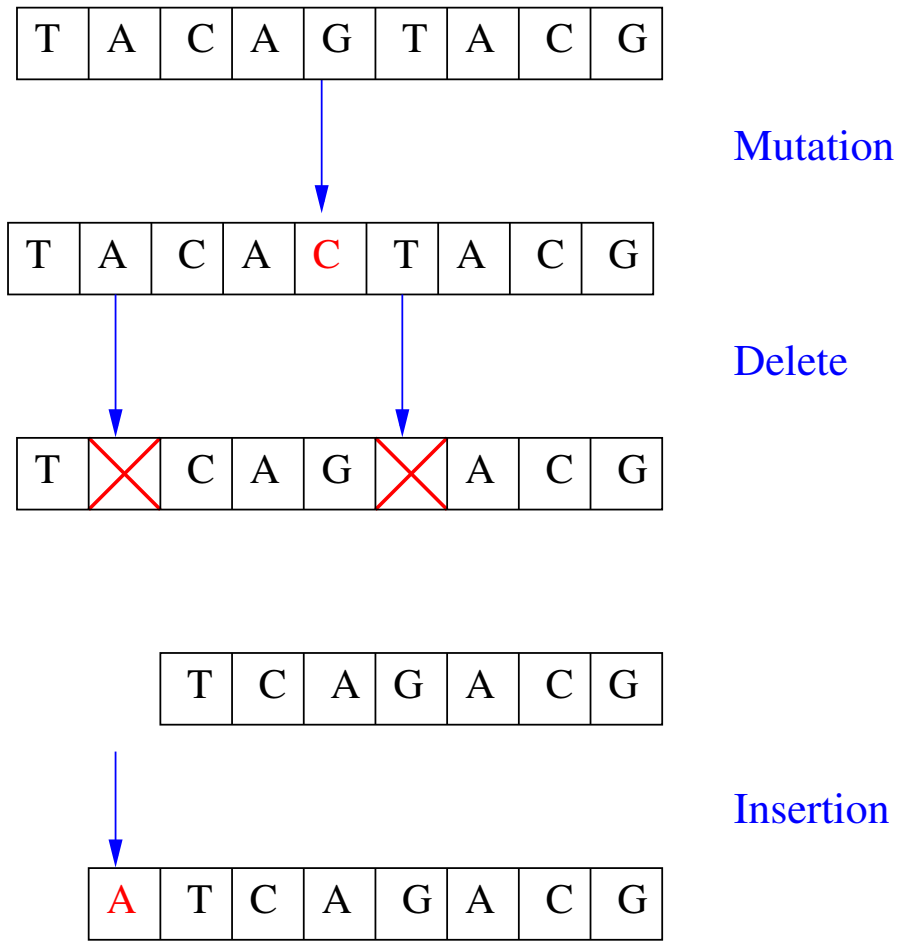


- The nucleus of each cell contains the DNA molecule, packaged into thread-like structures called chromosomes. In humans, each cell normally contains 23 pairs of chromosomes, for a total of 46.
- A gene is the basic unit of heredity, which are made up of DNA, and act as instructions to make proteins. Humans, have between 20,000 and 25,000 genes.
- Every person has 2 copies of each gene, one i from each parent. Most genes are the same in all humans, but 0.1% of genes are slightly different between people. Alleles are forms of the same gene with small differences in their sequence of DNA. These small differences contribute to each person's unique traits.

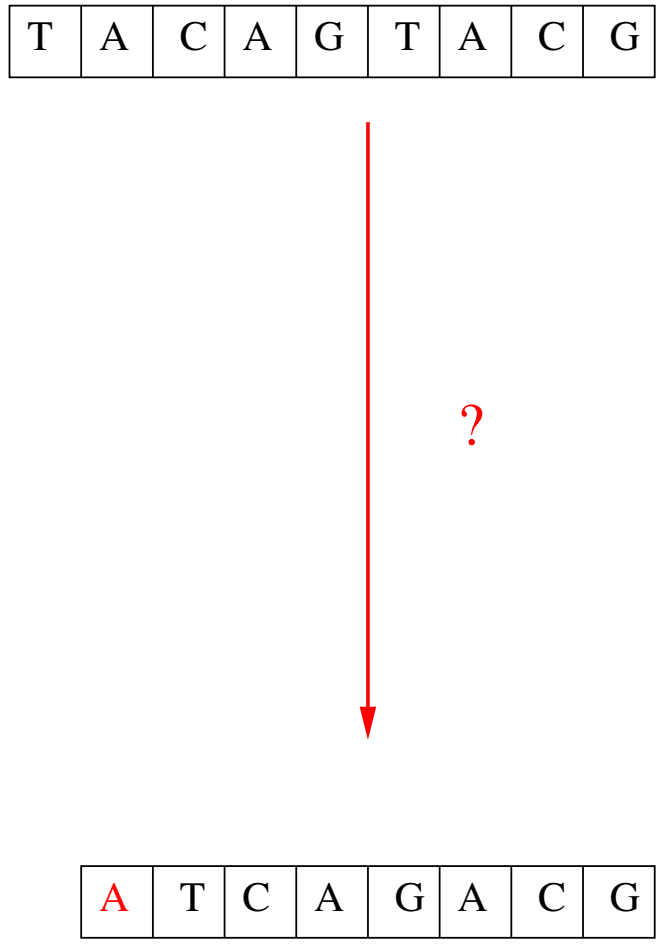
Computational genomics: Some questions

- When a new gene is discovered, one way to gain insight into its working, is to find well known genes (not necessarily in the same species) which match it closely. Biologists suggest a generalization of edit distance as a definition of approximately match.
- GenBank (`https://www.ncbi.nlm.nih.gov/genbank/`) has a collection of $> 10^{10}$ well studied genes, BLAST is a software to do fast searching for similarities between a genes a DB of genes.
- Sequencing DNA: consists in the determination of the order of DNA bases, in a short sequence of 500-700 characters of DNA. To get the global picture of the whole DNA chain, we generate a large amount of DNA sequences and try to assembled them into a coherent DNA sequence. This last part is usually a difficult one, as the position of each sequence is the global DNA chain is not know before hand.

Evolution DNA

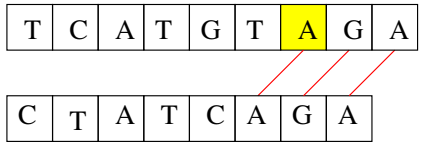


Sequence alignment problem

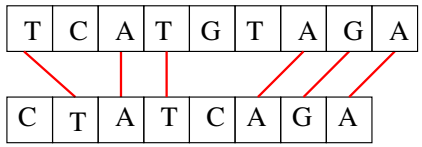


Formalizing the problem

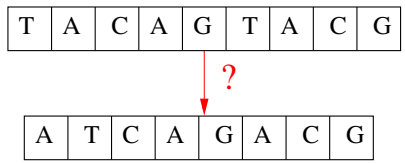
Longest common substring: Substring = chain of characters without gaps.



Longest common subsequence: Subsequence = ordered chain of characters with gaps.



Edit distance: Convert one string into another one using a given set of operations.



String similarity problem: The Longest Common Subsequence

LCS
INPUT: sequences $X =< x_1 \cdots x_m >$ and $Y =< y_1 \cdots y_n >$
QUESTION: Compute the longest common subsequence.

A sequence $Z =< z_1 \cdots z_k >$ is a *subsequence* of X if there is a subsequence of integers $1 \leq i_1 < i_2 < \ldots < i_k \leq m$ such that $z_j = x_{i_j}$. If Z is a subsequence of X and Y, the Z is a common subsequence of X and Y.

Given $X = ATATAT$, then TTT is a subsequence of X

Greedy approach

LCS: Given sequences $X =< x_1 \cdots x_m >$ and $Y =< y_1 \cdots y_n >$. Compute the longest common subsequence.

Greedy X, Y
$S := \emptyset$
for $i = 1$ to m **do**
  for $j = i$ to n **do**
    if $x_i = y_j$ **then**
      $S := S \cup \{x_i\}$
    end if
    let y_l such that $l = \min\{a > j | x_i = y_a\}$
    let x_k such that $k = \min\{a > i | x_i = y_a\}$
    if $\exists i, l < k$ **then**
      do $S := S \cup \{x_i\}$, $i := i + 1$;
      $j := l + 1$
      $S := S \cup \{x_k\}$, $i := k + 1$; $j := j + 1$
    else if not such y_l, x_k **then**
      do $i := i + 1$ and $j := j + 1$.
    end if
  end for
end for

Greedy approach

LCS: Given sequences $X =< x_1 \cdots x_m >$ and $Y =< y_1 \cdots y_n >$. Compute the longest common subsequence.

Greedy X, Y
$S := \emptyset$
for $i = 1$ to m **do**
  for $j = i$ to n **do**
    if $x_i = y_j$ **then**
      $S := S \cup \{x_i\}$
    end if
    let y_l such that $l = \min\{a > j | x_i = y_a\}$
    let x_k such that $k = \min\{a > i | x_i = y_a\}$
    if $\exists i, l < k$ **then**
      do $S := S \cup \{x_i\}$, $i := i + 1$;
      $j := l + 1$
      $S := S \cup \{x_k\}$, $i := k + 1$; $j := j + 1$
    else if not such y_l, x_k **then**
      do $i := i + 1$ and $j := j + 1$.
    end if
  end for
end for

Greedy approach does not work

For X =A T C A C and Y =C G C A C A C A C T the result of greedy is A T but the solution is A C A C.

DP approach: Characterization of optimal solution

Let $X =< x_1 \cdots x_n >$ and $Y =< y_1 \cdots y_m >$.
Let $X[i] =< x_1 \cdots x_i >$ and $Y[i] =< y_1 \cdots y_j >$.

Define $c[i, j]$ = length de la LCS of $X[i]$ and $Y[j]$.

Want $c[n, m]$ i.e. solution LCS X and Y.

What is a subproblem?
Subproblem = something that goes part of the way in converting one string into other.

Characterization of optimal solution and recurrence

- If X =C G A T C and Y =A T A C $c[5,4] = c[4,3] + 1$
- If X =C G A T and Y =A T A to find $c[4,3]$:
  - either LCS of C G A T and A T
  - or LCS of C G A and A T A

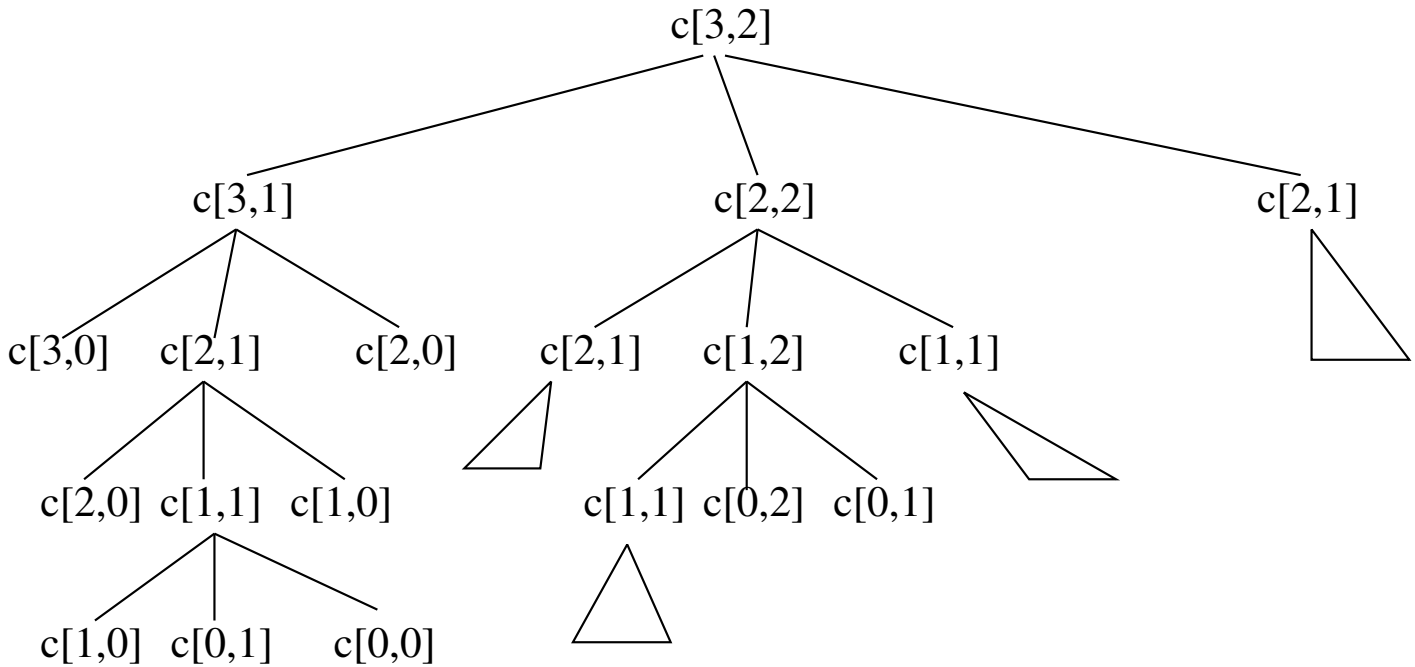
  $c[4,3] = \max(c[3,3], c[4,2])$

Therefore, given X and Y

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

Recursion tree

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$



The direct top-down implementation of the recurrence

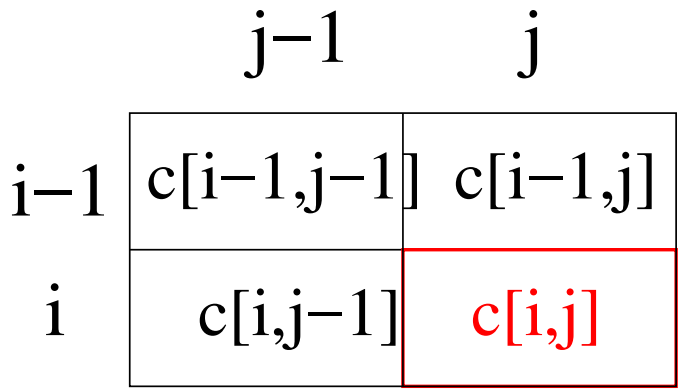
LCS (X, Y)
if $m = 0$ or $n = 0$ **then**
  return 0
else if $x_m = y_n$ **then**
  return $1+$**LCS** $(x_1 \cdots x_{m-1}, y_1 \cdots y_{n-1})$
else
  return $\max\{$**LCS** $(x_1 \cdots x_{m-1}, y_1 \cdots y_n)$
end if
LCS $(x_1 \cdots x_m, y_1 \cdots y_{n-1})\}$

The algorithm explores a tree of depth $\Theta(n+m)$, therefore the time complexity is $T(n) = 3^{\Theta(n+m)}$.

Bottom-up solution

Avoid the exponential running time, by tabulating the subproblems and not repeating their computation.
To memoize the values $c[i, j]$ we use a table $c[0 \cdots n, 0 \cdots m]$

Starting from $c[0, j] = 0$ for $0 \leq j \leq m$ and from $c[i, 0] = 0$ from $0 \leq i \leq n$ go filling row-by-row, left-to-right, all $c[i, j]$



Use a field $d[i, j]$ inside $c[i, j]$ to indicate from where we use the solution.

Bottom-up solution

LCS (X, Y)
for $i = 1$ **to** n **do**
  $c[i, 0] := 0$
end for
for $j = 1$ **to** m **do**
  $c[0, j] := 0$
end for
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    if $x_i = y_j$ **then**
      $c[i, j] := c[i-1, j-1] + 1$, $b[i.j] :=\nwarrow$
    else if $c[i-1, j] \geq c[i, j-1]$ **then**
      $c[i, j] := c[i-1, j]$, $b[i, j] :=\leftarrow$
    else
      $c[i, j] := c[i, j-1]$, $b[i, j] :=\uparrow$.
    end if
  end for
end for

Time and space complexity $T = O(nm)$.

Example.

X=(ATCTGAT); Y=(TGCATA). Therefore, $m = 6, n = 7$

		0	1 T	2 G	3 C	4 A	5 T	6 A
0		0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	T	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	T	0	↖1	↑1	↑2	↑2	↖3	←3
5	G	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	T	0	↖1	↑2	↑2	↑3	4	↑4

Construct the solution

Uses as input the table $c[n, m]$.
The first call to the algorithm is **con-LCS** (c, n, m)

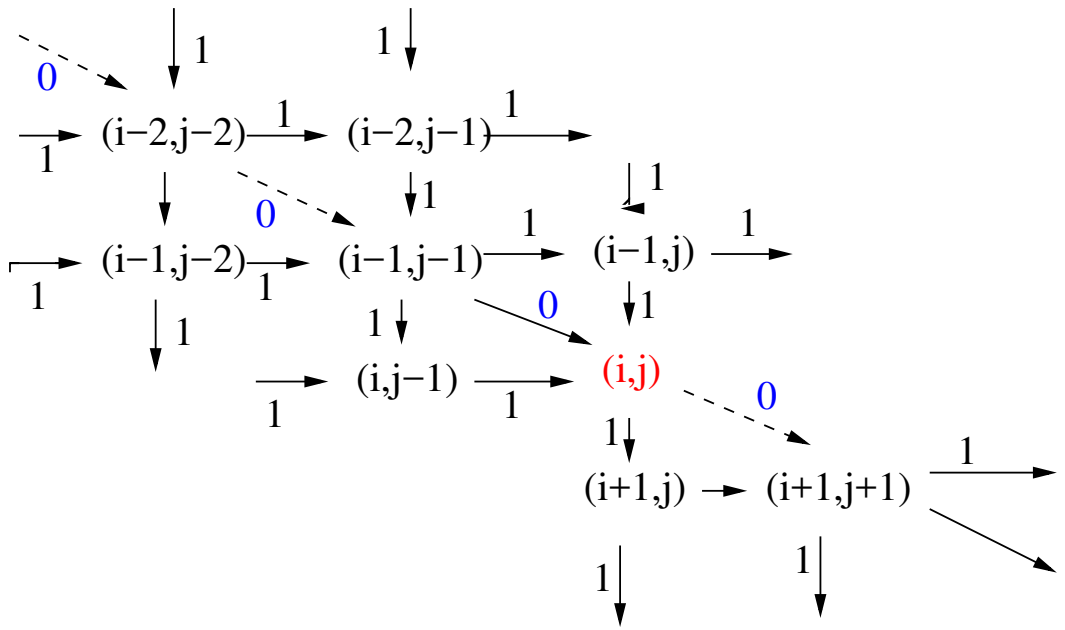
con-LCS (c, i, j)
if $i = 0$ or $j = 0$ **then**
  STOP.
else if $b[i, j] = \nwarrow$ **then**
  con-LCS $(c, i - 1, j - 1)$
  return x_i
else if $b[i, j] = \uparrow$ **then**
  con-LCS $(c, i - 1, j)$
else
  con-LCS $(c, i, j - 1)$
end if

The algorithm has time complexity $O(n + m)$.

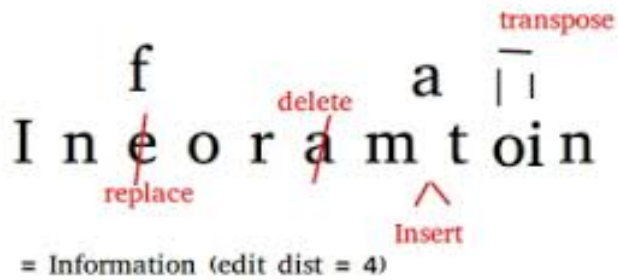
LCS: Underlying DAG

Assign weights 0 at edges $(i-1, j-1) \rightarrow (i, j)$ and 1 to remaining edges in the DAG

Find the minimum path between $(0, 0) \rightarrow (n, W)$



Edit Distance.



The edit distance between strings $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ is defined to be the *minimum* number of *edit operations* needed to transform X into Y.

Edit Distance: Levenshtein distance

The most relevant set of operations used are given by the Levenshtein distance:

- insert$(X, i, a) = x_1 \cdots x_i a x_{i+1} \cdots x_n$.
- delete$(X, i) = x_1 \cdots x_{i-1} x_{i+1} \cdots x_n$
- modify$(X, i, a) = x_1 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

The cost of each operation is 1.

Main applications of the Levenshtein distance:

- Computational genomics: similarity between strings on $\{A, T, G, C, -\}$.
- Natural Language Processing: distance, between strings on the alphabet.

Exemple-1

$x = aabab$ and $y = babb$

$aabab = X$

$X' =$insert$(X, 0, b)$ $baabab$

$X'' =$delete$(X', 2)$ $babab$

$Y =$delete$(X'', 4)$ $babb$

$X = aabab \rightarrow Y = babb$

Exemple-1

$x = aabab$ and $y = babb$

$aabab = X$

$X' =$insert$(X, 0, b)$ $baabab$

$X'' =$delete$(X', 2)$ $babab$

$Y =$delete$(X'', 4)$ $babb$

$X = aabab \rightarrow Y = babb$

A shortest edit distance

$aabab = X$

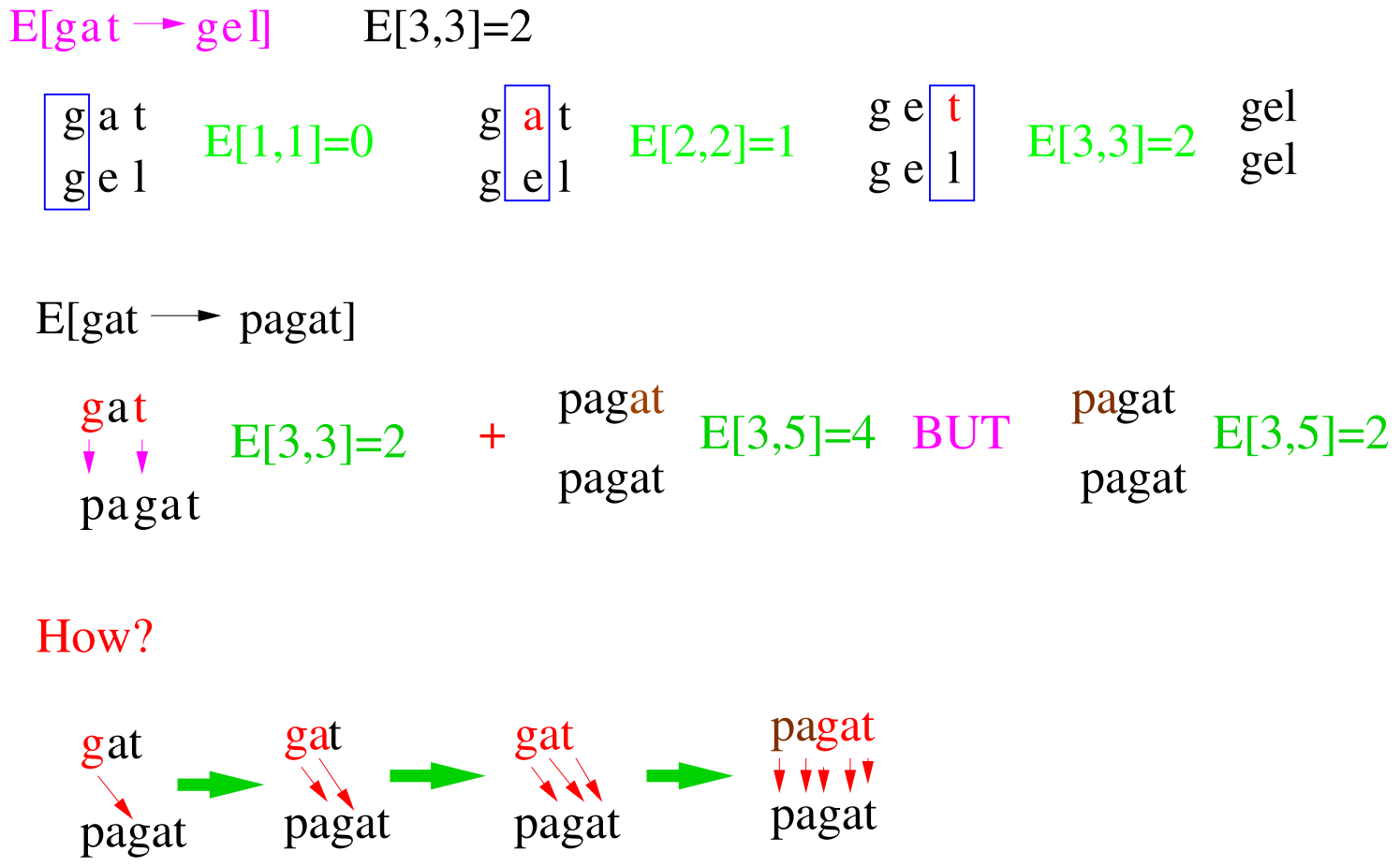
$X' =$modify$(X, 1, b)$ $babab$

$Y =$delete$(X', 4)$ $babb$

Use dynamic programming.

Exemple-2

Let $E[n, m]$ be the minimum Levenshtein distance between chains with length n and m respectively,



1.- Characterize the structure of an optimal solution and set the recurrence.

Assume want to find the edit distance from $X = TATGCAAGTA$ to $Y = CAGTAGTC$.

Let $E[10, 8]$ be the min. distance between X and Y.

Consider the prefixes $TATGCA$ and $CAGT$

Let $E[6, 4]$ = edit distance between $TATGCA$ and $CAGT$

- Distance between $TATGCA$ and $CAGT$ is $E[5, 4] + 1$ (delete A in X D)
- Distance between $TATGCAT$ and $CAGT$ is $E[6, 3] + 1$ (insert T in X I)
- Distance between $TATGCA$ and $CAGT$ is $E[5, 3] + 1$ (modify A in X to a T (M))

Consider the prefixes $TATGCA$ and $CAGTA$

- Distance between $TATGCA$ and $CAGTA$ is $E[5, 4]$ (keep last A and compare $TATGC$ and $CAGT$).

To compute the edit distance from $X = x_1 \cdots x_n$ to $Y = y_1 \cdots y_m$

Let $X[i] = x_1 \cdots x_i$ and $Y[j] = y_1 \cdots y_j$
let $E[i,j]$ = edit distance from $X[i]$ to $Y[j]$
If $x_i \neq y_j$ the last step from $X[i] \to Y[j]$ must be one of:

1. I *put y_j at the end x*: $x \to x_1 \cdots x_i y_j$, and then transform $x_1 \cdots x_i$ into $y_1 \cdots y_{j-1}$.

$$E[i,j] = E[i,j-1] + 1$$

2. D *delete x_i*: $x \to x_1 \cdots x_{i-1}$, and then transform $x_1 \cdots x_{i-1}$ into $y_1 \cdots y_j$.

$$E[i,j] = E[i-1,j] + 1$$

3. M *change x_i into y_j*: $x \to x_1 \cdots x_{i-1} y_j$, and then transform $x_1 \cdots x_{i-1}$ into $y_1 \cdots y_{j-1}$

$$E[i,j] = E[i-1,j-1] + 1$$

4. *if $x_i = y_j$*:

$$E[i,j] = E[i-1,j-1] + 0$$

Recurrence

Therefore, we have the recurrence

$$E[i,j] = \begin{cases} i & \text{if } j = 0 \text{ (converting } \lambda \to y[j]) \\ j & \text{if } i = 0 \text{ (converting } X[i] \to \lambda) \\ \min \begin{cases} E[i-1,j]+1 & \text{if D} \\ E[i,j-1]+1, & \text{if I} \\ E[i-1,j-1]+d(x_i,y_j) & \text{otherwise} \end{cases} \end{cases}$$

where

$$d(x_i,y_j) = \begin{cases} 0 & \text{if } x_i = y_j \\ 1 & \text{otherwise} \end{cases}$$

2.- Computing the optimal costs.

Edit $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$
for $i = 0$ **to** n **do**
  $E[i, 0] := i$
end for
for $j = 0$ **to** m **do**
  $E[0, j] := j$
end for
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    if $x_i = y_j$ **then**
      $d(x_i, y_j) = 0$
    else
      $d(x_i, y_j) = 1$
    end if
    $E[i, j] := E[i, j-1] + 1$
    if $E[i-1, j-1] + d(x_i, y_j) < E[i, j]$
    then
      $E[i, j] := E[i-1, j-1] + d(x_i, y_j)$,
      $b[i, j] :=\nwarrow$
    else if $E[i-1, j] + 1 < E[i, j]$ **then**
      $E[i, j] := E[i-1, j] + 1$, $b[i, j] :=\uparrow$
    else
      $b[i, j] :=\leftarrow$
    end if
  end for
end for

Time and space complexity $T = O(nm)$.

Example:

X=aabab; Y=babb. Therefore,

$n = 5, m = 4$

		0	1	2	3	4
		λ	b	a	b	b
0	λ	0	1	2	3	4
1	a	1	$\nwarrow$ 1	$\nwarrow$ 1	$\leftarrow$ 2	$\leftarrow$ 3
2	a	2	$\nwarrow$ 2	$\nwarrow$ 1	$\nwarrow$ 2	$\nwarrow$ 3
3	b	3	$\nwarrow$ 2	$\uparrow$ 2	$\nwarrow$ 1	$\nwarrow$ 2
4	a	4	$\uparrow$ 3	$\nwarrow$ 2	$\uparrow$ 2	$\nwarrow$ 2
5	b	5	$\nwarrow$ 4	$\uparrow$ 3	$\uparrow$ 2	$\nwarrow$ 2

3.- Construct the solution.

Uses as input the table $E[n, m]$.
The first call to the algorithm is **con-Edit** (E, n, m)

con-Edit (E, i, j)
if $i = 0$ or $j = 0$ **then**
  STOP.
else if $b[i, j] = \nwarrow$ and $x_i = y_j$ **then**
  change(X, i, y_j)
  con-Edit $(E, i - 1, j - 1)$
else if $b[i, j] = \uparrow$ **then**
  delete(X, i) , **con-Edit** $(c, i - 1, j)$
else
  insert(X, i, y_j), **con-Edit** $(c, i, j - 1)$
end if

This algorithm has time complexity $O(nm)$.

Sequence Alignment.

Finding similarities between sequences is important in Bioinformatics

For example,

- Locate similar subsequences in DNA
- Locate DNA which may overlap.

Similar sequences evolved from common ancestor

Evolution modified sequences by mutations

- Replacement.
- Deletions.
- Insertions.

Sequence Alignment.

Given two sequences over same alphabet

G C G C A T G G A T T G A G C G A

T G C G C C A T T G A T G A C C A

An *alignment*

- GCGC- ATGGATTGAGCGA

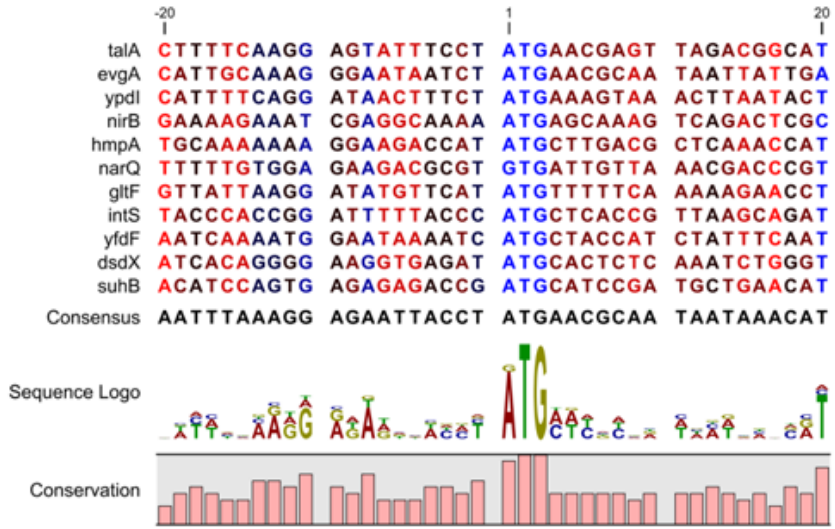
TGCGCCATTGAT -GACC- A

Alignments consist of:

- Perfect matchings.
- Mismatches.
- Insertion and deletions.

There are many alignments of two sequences.

Which is better?



PD for sequence alignment

When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches.
The closeness of two genes is measured by the extent to which they are aligned.

For example, consider two genes X =ATGC and Y =TACGCA.
An alignment of x and y is a way of matching up these two strings by writing them in columns:

$$\begin{array}{cccccc} - & A & T & G & C & - \\ T & A & C & G & C & A \end{array}$$

The score of an alignment is given by a matrix δ of size $(|\Sigma| + 1) \times (|\Sigma| + 1)$.

PD for sequence alignment

For instance the previous alignment has score:

$$\delta(-, T) + \delta(A, A) + \delta(T, C) + \delta(G, G) + \delta(C, C) + \delta(-, A),$$

and we could have chosen the score matrix

$$\begin{array}{c} \begin{array}{ccccc} A & T & C & G & - \end{array} \\ \begin{pmatrix} +1 & -1 & -1 & -1 & 0 \\ -1 & +1 & -1 & -1 & 0 \\ -1 & -1 & +1 & -1 & 0 \\ -1 & -1 & -1 & +1 & 0 \\ 0 & 0 & 0 & 0 & \end{pmatrix} \end{array} \begin{array}{c} A \\ T \\ C \\ G \\ - \end{array}$$

So for this δ the proposed alignment has a score $= 2$.

PD for sequence alignment

An simple extension of the previous PD to solve the Edit distance can be used to solve the problem:

Given as input sequences x and y of DNA and an score matrix δ, output the alignment of x and y with maximum score.,

In this case, the recurrence is given by:

$$E[i,j] = \max\{E[i-1,j]+\delta(x_i,-), E[i-1,j-1]+\delta(x_i,y_j), E[i,j-1]+\delta(-,y_j),$$

with initial conditions $\forall i,j > 0$,
$E[0,0] = 0$; $E[i,0] = E[i-1,0] + \delta(x_i,-)$; and
$E[0,j] = E[0,j-1] + \delta(-,y_j)$.

Complexity: $O(nm)$.

Dynamic Programming in Trees

Trees are nice graphs to bound the number of subproblems.
Given $T = (V, A)$ with $|V| = n$, recall that there are n subtrees in T.

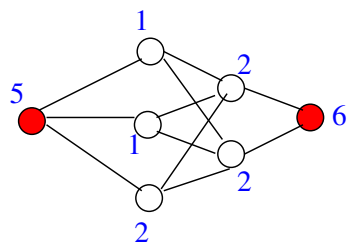
Therefore, *when considering problems defined on trees, it is easy to bound the number of subproblems*

This allows to use Dynamic Programming to give polynomial solutions to "difficult" graph problems when the input is a tree.

The Maximum Weight Independent Set (MWIS)

INPUT: $G = (V, E)$, together with a weight $w : V \to \mathbb{R}$
QUESTION: Find the largest $S \subseteq V$ such that no two vertices in S are connected in G.

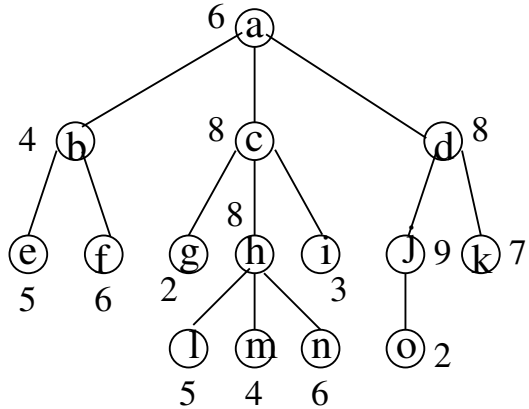
For general G, the problem is difficult, as the case with all weights $=1$ is already NP-complete.



MWIS on Trees

Given a tree $T = (V, E)$ choose a $r \in V$ and root it from r

INSTANCE: Given a rooted tree $T_r = (V, E)$ and a set of weights $w : V \to \mathbb{R}$,
QUESTION: Find the independent set of nodes with maximal weight.



Notation:

- Given T_r, where r is the root, then $\forall v \in V$, let T_v denote subtree rooted at v.
- Given $v \in T_r$ let $F(v)$ be the set of children of v, and $N(v)$ be the set of grandchildren of v.
- For any T_v, let S_v be the set of the MWIS in T_v, and let $M(v) = \sum_{x \in S_v} w(x)$. We want to max $M(r)$.

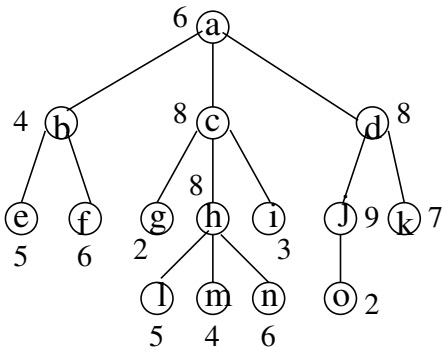
Characterization of the optimal solution

Key observation: An MWIS optima set S_r in T_r can't contain vertices which are father-son.

- If $r \in S_r$: then $F(r) \not\subseteq S_r$. So $S_r - \{r\}$ contains an optimum solution for each T_v, with $v \in N(r)$.
- If $r \notin S_r$: S_r contains an optimum solution for each T_u, with $u \in F(r)$.

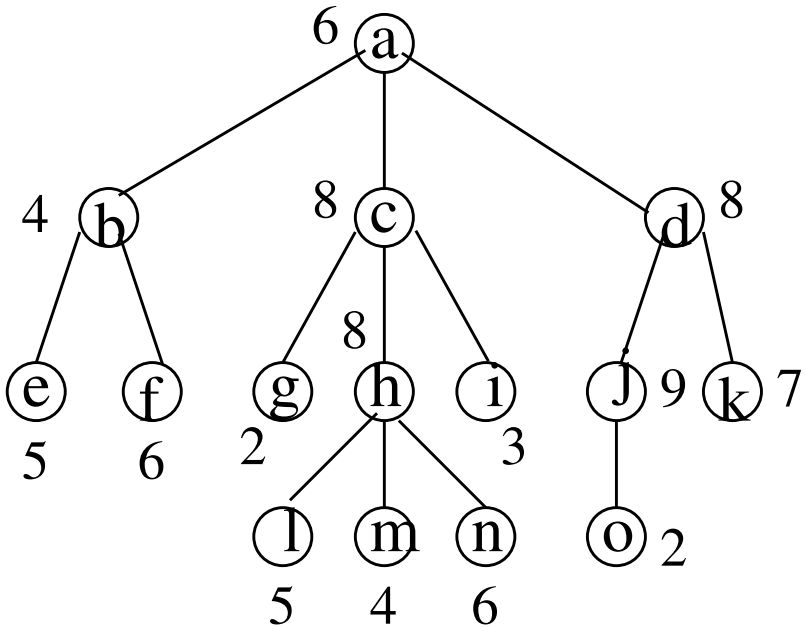
Recursive definition of the optimal solution

$$M(v) = \begin{cases} w(v) & \text{if } v \text{ is a leaf,} \\ \max\{\sum_{u \in F(v)} M(u), w(v) + \sum_{u \in N(v)} M(u)\} & \text{otherwise.} \end{cases}$$



For $v \in T$, define:
$M'(v) = \sum_{u \in N(v)} M(u)$
$M(v) = \max\{\sum_{u \in F(v)} M(u), w(v) + M'(v)\}$

Recall:Post-order traversal of a rooted tree



Post−Order

e f b g l m n h i c o j k d a

DP Algorithm to find M_r

Let $v_1, \ldots, v_n = r$ be the post-order traversal of T_r
Define a $2 \times n$ table A to store the values of M and M'.

WIS T_r
Let $v_1, \ldots, v_n = r$ the post-order traversal of T_r
for $i = 1$ **to** n **do**
if v_i a leaf **then**
$M(v_i) = w(v_i), M'(v_i) = 0$
else
$M'(v_i) = \sum_{u \in N(v_i)} M(u)$
$M(v_i) = \max\{\sum_{u \in F(v_i)} M(u), w(v_i) + M'(v_i)\}$
end if
store $M(v_i)$ and $M'(v_i)$ in $A[2i-1]$ and $A[2i]$
end for

Complexity: space $= O(n)$, time $= O(n)$

Bottom-up

$M(l) = 5, M(m) = 4, M(n) = 6, M(e) = 5$

$M(f) = 6, M(o) = 2, M(k) = 7$

$M(b) = 4,\ M'(b) = 11$

$M(h) = 8,\ M'(h) = 15$

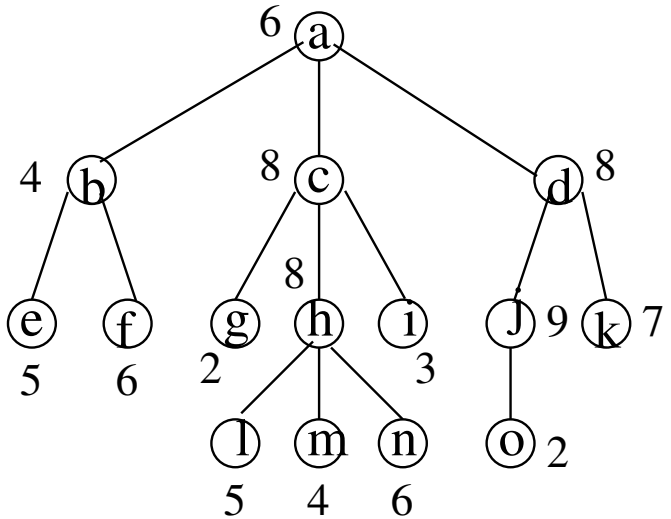
$M(j) = 9,\ M'(j) = 2$

$M(d) = 10,\ M'(d) = 16$

$M(c) = 23,\ M'(c) = 20$

$M(a) = 6 + M'(b) + M'(c) + M'(d) = 53$

$M'(a) = 6 + M'(b) + M(c) + M'(d) = 50$



	e	f	b	g	l	m	n	h	i	c	o	j	k	d	a
M	5	6	11	2	5	4	6	15	3	23	2	9	7	16	53
M'	0	0	0	0	0	0	0	0	0	15	0	0	0	2	50

How can we recover the set S_r??