

LCS- "ACDA"

Fibonacci Recurrence.

n -th Fibonacci Term

INPUT: $n \in \mathbb{N}$

QUESTION: Compute

$$F_n = F_{n-1} + F_{n-2}$$

Recursive Fibonacci (n)

if $n = 0$ then

return 0

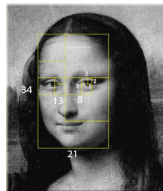
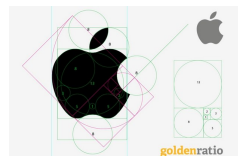
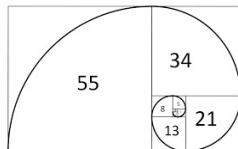
else if $n = 1$ then

return 1

else

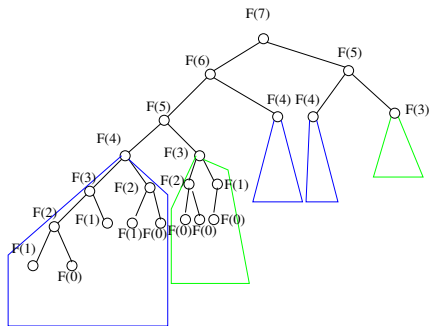
(Fibonacci($n - 1$)+
Fibonacci($n - 2$))

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}

$$F_0 = 0$$

$$F_1 = 1$$

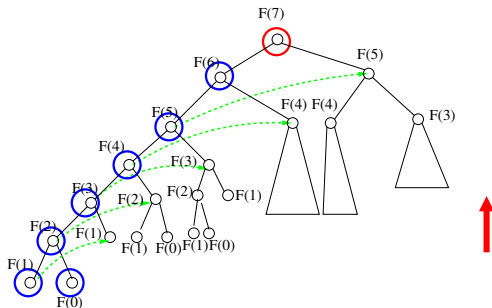
for $i = 1$ to n **do**

$$F_i = F_{i-1} + F_{i-2}$$

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

The dynamic programming technique is an exhaustive search method. In general those brute force methods are intractable for large inputs.

However, we can apply DP if the problem has a certain amount of structure that we can exploit: Dynamic Programming works when:

- ▶ **Simple Subproblems:** There must be a way of breaking the global optimization problem into subproblems, each having a similar structure to the original problem.
- ▶ **Optimal sub-structure:** An optimal solution to a problem must be a composition of optimal subproblem solutions, using a relatively simple combining operation.
- ▶ **Overlapping subproblems:** A recursive solution contains a small number of distinct subproblems, repeated many times.

This last property allows us to take advantage of **memoization**, avoid repeated recursive calls by storing intermediate values, in the appropriate data structure.

Difference with greedy

- ▶ Mainly optimization problems, which have **simple subproblems** with a similar structure to the original
- ▶ 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.

Implementation 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

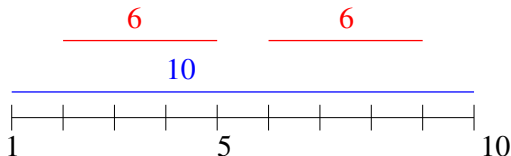
- ▶ The other way to implement DP is using **iterative algorithms**.
- ▶ DP is a trade-off between time speed vs. storage space.
- ▶ In general, although recursive algorithms have exactly the same running time than the iterative version, the constant factor in the O is quite more larger because the overhead 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: Given a set $S = \{1, 2, \dots, n\}$ 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 . Find the set of mutually compatible such that it maximizes $\sum_{i \in S} w_i$

Recall: Greedy strategy not always solved this problem.



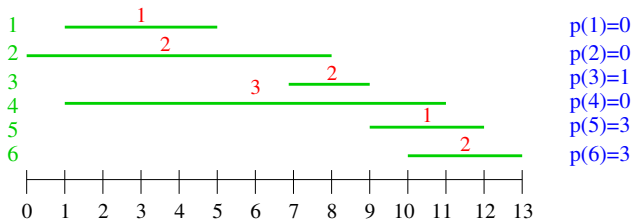
Notation for the WEIGHTED ACTIVITY SELECTION

We have $\{1, 2, \dots, n\}$ activities with $f_1 \leq f_2 \leq \dots \leq f_n$ 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 disjoint ($p(j) = 0$ if no disjoint $i < j$ exists).

Let $\text{Opt}(j)$ 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 $\{1, \dots, j\}$.



Recurrence

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

1.- $j \in O_j$:

- ▶ w_j is part of the solution,
- ▶ no jobs $\{p(j) + 1, \dots, j - 1\}$ are in O_j ,
- ▶ if $O_{p(n)}$ is the optimal solution for $\{1, \dots, p(n)\}$ 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 $P[1, \dots, n] = \{p[j]\}$.

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 $P[1 \dots n]$

To compute $\text{Opt}(j)$:

R-Opt (j)

if $j = 0$ **then**

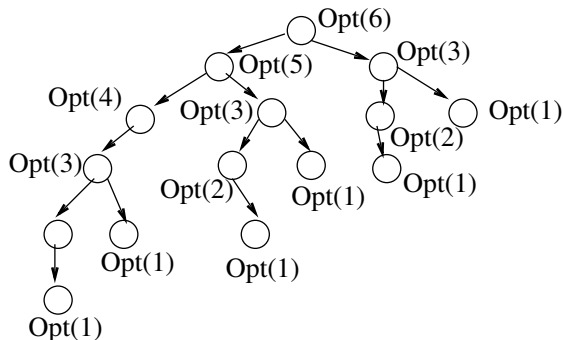
return 0

else

return $\max(w_j + \text{R-Opt}(p(j)), \text{R-Opt}(j - 1))$

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 $M[0, 1 \dots, n]$, 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**

$M[j] = \max(M[P[j]] + w_j, M[j - 1])$

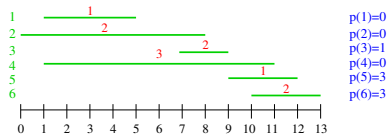
end for

return $M[n]$

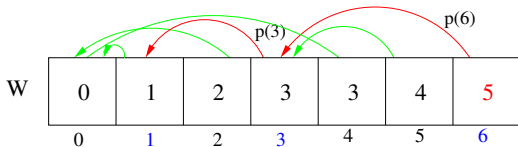
Notice: this algorithm gives only the numerical max. weight

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: $O(n)$

MULTIPLYING A SEQUENCE OF MATRICES

MULTIPLICATION OF n MATRICES Given as input a sequence of n matrices $(A_1 \times A_2 \times \cdots \times A_n)$. 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:

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

MULTIPLYING A SEQUENCE OF MATRICES

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, \dots, 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, \dots, 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.

MULTIPLYING A SEQUENCE OF MATRICES

How many ways to parenthesize A_1, \dots, 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 parenthesize A_1, \dots, A_n . Then,

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

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

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,

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

1.- Recurrence

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

That is,

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

2.- Computing the optimal costs

Straightforward implementation of the previous recurrence:

As $\dim(A_i) = p_{i-1}p_i$, the input is given by $P = \langle p_0, p_1, \dots, p_n \rangle$,

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

Complexity: $T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n \sim \Omega(2^n)$.

Dynamic programming approach

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

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**

$q = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$

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 .

$T(n) = \Theta(n^3),$
 $\text{space} = \Theta(n^2).$

Example.

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

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

$$l = 2, i = 1, j = 2,$$

$$k = 1 : q = m[1, 2] = m[1, 1] + m[2, 2] + 3.5.3 = 45 \text{ (} A_1 A_2 \text{)}$$

$$s[i, 2] = 1$$

$$l = 2, i = 2, j = 3,$$

$$k = 2 : q = m[2, 3] = m[2, 2] + m[3, 3] + 5.3.2 = 30 \text{ (} A_2 A_3 \text{)}$$

$$s[2, 3] = 2$$

$$l = 2, i = 3, j = 4,$$

$$k = 3 : q = m[3, 4] = m[3, 3] + m[4, 4] + 3.2.4 = 24 \text{ (} A_3 A_4 \text{)}$$

$$s[3, 4] = 3$$

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

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

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

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

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

$$s[2, 4] = 3$$

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

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

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

$i \setminus j$	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 $s[1, \dots, n, 1, \dots, n]$. In the table, $s[i, j]$ contains k such that the optimal way to multiply:

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

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, $A_{1-n} = A_{1-s[1, n]}A_{s[1, n] + 1 - n}$.

frametitle3.- Constructing an optimal solution

Multiplication(A, s, i, j)

if $j > 1$ **then**

$X = \text{Multiplication}(A, s, i, s[i, j])$

$Y := \text{Multiplication}(A, s, s[i, j] + 1, j)$

return $X \times Y$

else

return A_i

end if

Therefore $(A_1(A_2A_3))A_4$.

0-1 KNAPSACK

0-1 KNAPSACK: Given as 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 objects that yield that value.

Let $v[i, w]$ be the maximum value (optimum) we can get from objects $\{1, 2, \dots, i\}$ 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 keep $v[i - 1, w]$,
- ▶ or that the i -th element is part of the solution. we add v_i subtract 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 $M = v[0 \dots n, 0 \dots W]$,

```
Knapsack( $i, w$ )  
for  $i = 0$  to  $n$  do  
     $v[i, 0] = 0$   
end for  
for  $w = 1$  to  $W$  do  
     $v[0, w] = 0$   
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

$W = 11$.

		0	1	2	3	4	5	w 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,

$$v[4, 10] = \max\{v[3, 10], v[3, 10 - 6] + 22\} = \max\{25, 7 + 22\} = 29.$$

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

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

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

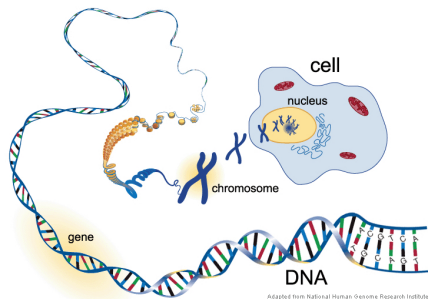
Hidden structure in a DP algorithm

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.

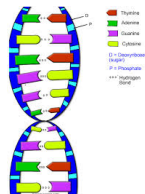
Having nodes a_1, a_2, \dots, a_n point to b means that b can be solved only when a_1, a_2, \dots, a_n are solved.

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

DNA: The book of life

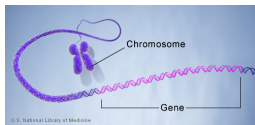


Adapted from National Human Genome Research Institute



- ▶ 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 in the global DNA chain is not know before hand.

Evolution DNA

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



Mutation

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



Delete

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

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



Insertion

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

Sequence alignment problem

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



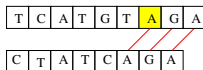
?

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

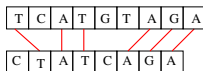
```
X: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
    | |  | | | | | | | | | | | |
    G TC  GT CG G AAGCCGGCCGAA
    GTCGT CGGAA GCCG  GC C G AA
    | | | | | | | | | | | | | |
Y:  GTCGTTCGGAATGCCGTTGCTCTGTAA
```

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 $X = \langle x_1 \cdots x_m \rangle$ and $Y = \langle y_1 \cdots y_n \rangle$

QUESTION: Compute the longest common subsequence.

A sequence $Z = \langle z_1 \cdots z_k \rangle$ is a *subsequence* of X if there is a subsequence of integers $1 \leq i_1 < i_2 < \dots < i_k \leq m$ 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 $X = \langle x_1 \cdots x_m \rangle$ and $Y = \langle y_1 \cdots y_n \rangle$.
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 $l = \min\{a > j \mid x_i = y_a\}$

 let x_k such that $k = \min\{a > i \mid x_a = y_j\}$

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 $X = \langle x_1 \cdots x_m \rangle$ and $Y = \langle y_1 \cdots y_n \rangle$.
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 $X = \langle x_1 \cdots x_n \rangle$ and $Y = \langle y_1 \cdots y_m \rangle$.

Let $X[i] = \langle x_1 \cdots x_i \rangle$ and $Y[j] = \langle y_1 \cdots y_j \rangle$.

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 $X = \text{C G A T C}$ and $Y = \text{A T A C}$ $c[5, 4] = c[4, 3] + 1$
- ▶ If $X = \text{C G A T}$ and $Y = \text{A T A}$ 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

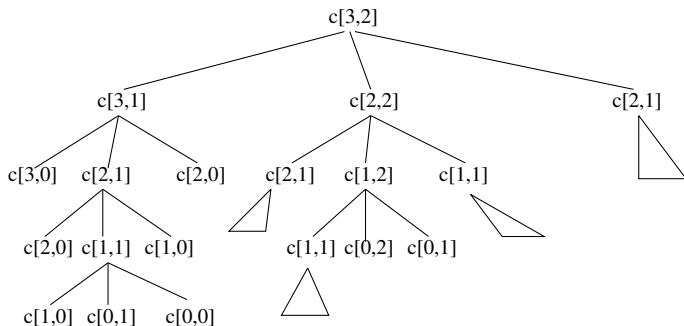
$$c[4, 3] = \max(c[3, 3], c[4, 2])$$

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  $1 + \text{LCS}(x_1 \cdots x_{m-1}, y_1 \cdots y_{n-1})$   
else  
    return  $\max\{\text{LCS}(x_1 \cdots x_{m-1}, y_1 \cdots y_n),$   
             $\text{LCS}(x_1 \cdots x_m, y_1 \cdots y_{n-1})\}$   
end if
```

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 $c[i, j]$

	j-1	j
i-1	$c[i-1, j-1]$	$c[i-1, j]$
i	$c[i, j-1]$	$c[i, j]$

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
```

complexity:
 $T = O(nm)$.

Example.

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

		0	1	2	3	4	5	6
			T	G	C	A	T	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

Following the arrows: TCTA

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 ($c, i, j - 1$)

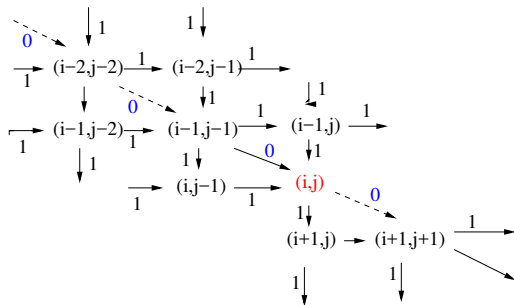
end if

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 $(0, 0) \rightarrow (n, W)$



The EDIT DISTANCE problem

Information

edit dist = 4

Operations: replace, delete, insert, transpose

= Information (edit dist = 4)

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 .

All the operations are done on X

EDIT DISTANCE: Levenshtein distance

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

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

In the Levenshtein distance the cost of modify is 2, and the cost of insert/delete is 1.

To simplify we assume that *the cost of each operation is 1*.

Main applications of the Levenshtein distance:

- ▶ Computational genomics: evolution between generations, i.e. between strings on $\{A, T, G, C, -\}$.
- ▶ Natural Language Processing: distance, between strings on the alphabet.

Example-1

$X = aabab$ and $Y = babb$

$aabab = X$

$X' = \text{insert}(X, 0, b)$ *baabab*

$X'' = \text{delete}(X', 2)$ *babab*

$Y = \text{delete}(X'', 4)$ *babb*

$X = aabab \rightarrow Y = babb$

Example-1

$X = aabab$ and $Y = babb$

$aabab = X$

$X' = \text{insert}(X, 0, b)$ *baabab*

$X'' = \text{delete}(X', 2)$ *babab*

$Y = \text{delete}(X'', 4)$ *babb*

$X = aabab \rightarrow Y = babb$

A shortest edit distance

$aabab = X$

$X' = \text{modify}(X, 1, b)$ *babab*

$Y = \text{delete}(X', 4)$ *babb*

Use dynamic programming.

Example-2

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

$E[\text{gat} \rightarrow \text{gel}] \quad E[3,3]=2$

$\begin{bmatrix} \text{g} & \text{a} & \text{t} \\ \text{g} & \text{e} & \text{l} \end{bmatrix} \quad E[1,1]=0 \quad \begin{bmatrix} \text{g} & \text{a} & \text{t} \\ \text{g} & \text{e} & \text{l} \end{bmatrix} \quad E[2,2]=1 \quad \begin{bmatrix} \text{g} & \text{e} & \text{t} \\ \text{g} & \text{e} & \text{l} \end{bmatrix} \quad E[3,3]=2 \quad \begin{bmatrix} \text{g} & \text{e} & \text{l} \\ \text{g} & \text{e} & \text{l} \end{bmatrix}$

$E[\text{gat} \rightarrow \text{pagat}]$

$\begin{bmatrix} \text{g} & \text{a} & \text{t} \\ \downarrow & \downarrow & \\ \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \end{bmatrix} \quad E[3,3]=2 \quad + \quad \begin{bmatrix} \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \\ \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \end{bmatrix} \quad E[3,5]=4 \quad \text{BUT} \quad \begin{bmatrix} \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \\ \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \end{bmatrix} \quad E[3,5]=2$

How?

$\begin{bmatrix} \text{g} & \text{a} & \text{t} \\ \searrow & & \\ \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \end{bmatrix} \rightarrow \begin{bmatrix} \text{g} & \text{a} & \text{t} \\ \searrow & \searrow & \\ \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \end{bmatrix} \rightarrow \begin{bmatrix} \text{g} & \text{a} & \text{t} \\ \searrow & \searrow & \searrow & \\ \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \end{bmatrix} \rightarrow \begin{bmatrix} \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \text{p} & \text{a} & \text{g} & \text{a} & \text{t} \end{bmatrix}$

NOTICE: $E[\text{gat} \rightarrow \text{pagat}]$ equivalent to $E[\text{pagat} \rightarrow \text{gat}]$

1.- Characterize the structure of an optimal solution

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

Consider the prefixes $TATGCA$ and $CAGTA$

- Distance between $TATGC$ **A** and $CAGT$ **A** 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 $X[i] \rightarrow Y[j]$ must be one of:

1. **I** *put y_j at the end x* : $x \rightarrow x_1 \cdots x_i y_j$, 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* : $x \rightarrow x_1 \cdots x_{i-1}$, 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* : $x \rightarrow x_1 \cdots x_{i-1} y_j$, 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 $x_i = y_j$* :

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

1.- Recurrence

Therefore, we have the recurrence

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

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  $X = \{x_1, \dots, x_n\}$ ,  $Y = \{y_1, \dots, y_n\}$ 
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
             $E[i, j] = E[i - 1, j] + 1$ ,  $b[i, j] := \leftarrow$ 
        else
             $b[i, j] := \uparrow$ 
        end if
    end for
end for
```

Space and time complexity:

$O(nm)$.

\leftarrow is a **L** operation,
 \uparrow is a **D** operation, and
 \nwarrow is either a **M** or a
no-operation.

2.- Computing the optimal costs: Example

$X = \text{a a b a b}$; $Y = \text{b a b b}$. Therefore, $n = 5$, $m = 4$

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

\leftarrow is a **I** operation, \uparrow is a **D** operation, and

\nearrow is either a **M** or a **no-operation**.

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 problem

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 problem

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*

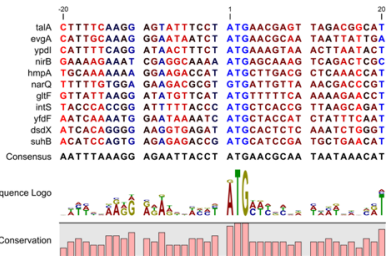
- GCGC- ATGGATTGAGCGA
TGCGCCATTGAT -GACC- A

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 $X = \text{ATGC}$ and $Y = \text{TACGCA}$. An **alignment** of x and y is a way of matching up these two strings by writing them in columns:

—	A	T	G	C	—
T	A	C	G	C	A

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

$$\begin{array}{ccccc} & A & T & C & G & - \\ \left(\begin{array}{ccccc} +1 & -1 & -1 & -1 & 0 \\ -1 & +1 & -1 & -1 & 0 \\ -1 & -1 & +1 & -1 & 0 \\ -1 & -1 & -1 & +1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right) & A & T & C & G & - \end{array}$$

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

PD for SEQUENCE ALIGNMENT

An simple modification of the previous PD 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*

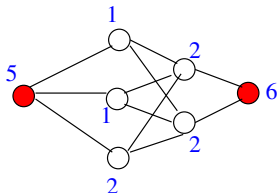
This allows to use Dynamic Programming to give polynomial solutions to "difficult" graph problems when the input is a tree.

The MAXIMUM WEIGHT INDEPENDENT SET (MWIS)

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

For general graphs, the problem is difficult, even for the case with weights 1

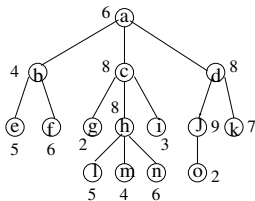
i.e. MAXIMUM INDEPENDENT SET is NP-complete.



MAXIMUM WEIGHT INDEPENDENT SET on Trees

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

i.e. Given a rooted tree $T_r = (V, E)$ and a set of weights $w : V \rightarrow \mathbb{R}$, 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 $u \in F(r)$.

For every subtree rooted at v , we consider two values,

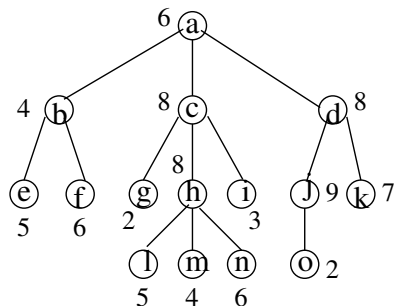
- ▶ $M(v)$: value of the solution if we take v as part of the solution,
- ▶ $M'(v)$: value of the solution if we DON'T take v as part of the solution.

Recursive definition of the optimal solution

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

Notice that for any $v \in T$: $M'(v) = \sum_{u \in N(v)} M(u)$.

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 $v_1, \dots, v_n = r$ 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 $v_1, \dots, v_n = r$ 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

$M'(v_i) = \sum_{u \in N(v)} M(u)$ (0 if $N(v) = \emptyset$)

$M(v_i) = \max\{\sum_{u \in F(v_i)} M(u), w(v_i) + M'(v_i)\}$

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

$M(e) = 5, M(f) = 6, M(b) = 11, M'(b) = 0$

$M(g) = 2, M(l) = 5, M(m) = 4, M(n) = 6,$

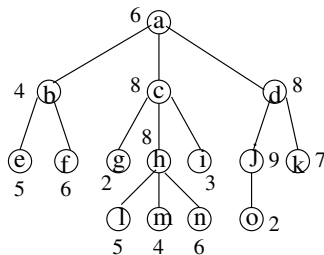
$M(h) = 15,$

$M(i) = 5, M'(c) = 15, M(c) = 23$

$M(o) = 2, M(j) = 9, M(k) = 7,$

$M(d) = 16, M'(d) = 2$

$M'(a) = 47, M(a) = 53$



	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	47

How can we recover the set S_r ??

Top-down algorithm to recover the WIS S_r

Let $r = v_n, \dots, v_1$ be the level-order traversal of T_r

Recover-WIS $T_r, A[2n]$

$S_r = \emptyset$

for $i = n$ **to** 1 **do**

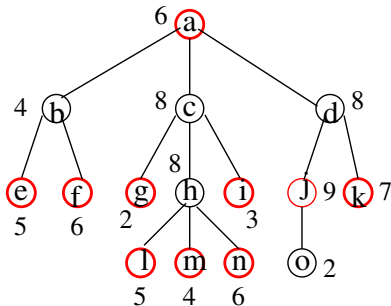
if $M(v_i) = M'(v_i) + w(v_i)$ **then**

$S = S \cup \{v_i\}$

end if

end for

return S



Maximum Weighted Independent Set:

$S = \{a, e, f, g, i, j, k, l, m, n\}$ $w(S) = 53$.

Complexity.

Space complexity: $2n^2$

Time complexity:

Step 1 To update M and M' we only need to look at the sons of the vertex into consideration. As we cover all n vertices, $O(n)$

Step 2 From the root down to the leaves, for each vertex v compare $M(v)$ with $M'(v)$, $\Theta(n)$

As we have to consider all the vertices at least one, we have a lower bound in the time complexity of $\Omega(n)$, therefore the total number of steps is $\Theta(n)$.