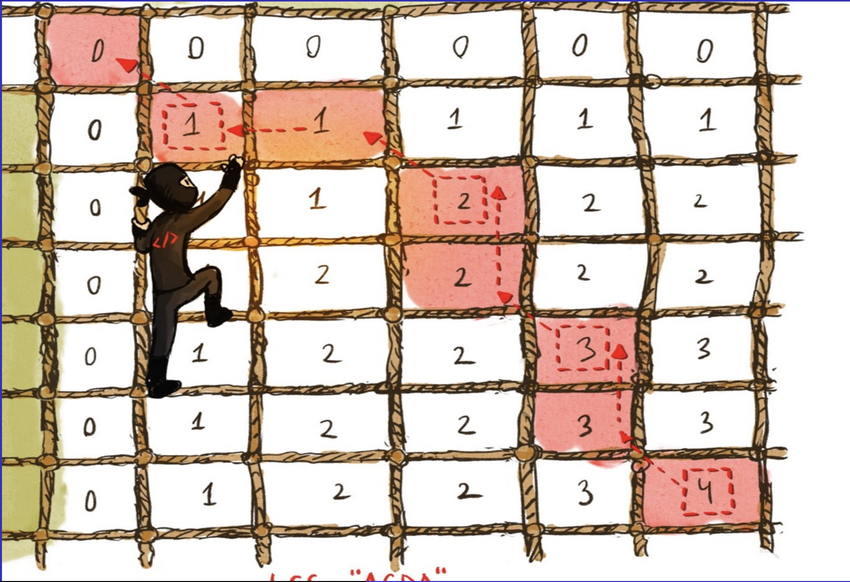
Dynamic Programming

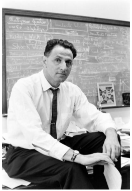


0	0	0	0	0	0
0	1	1	1	1	1
0		1	2	2	2
0		2	2	2	2
0	1	2	2	3	3
0	1	2	2	3	3
0	1	2	2	3	4

Dynamic Programming

For a gentle introduction to DP see Chapter 6 in DPV, KT and CLRS also have a chapter devoted to DP.

Richard Bellman: *An introduction to the theory of dynamic programming* RAND, 1953



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

Dynamic Programming

Dynamic Programming works efficiently when:

- Subproblems: There must be a way of breaking the global optimization problem into subproblems, each having a similar structure to the original problem but smaller size.
- Optimal sub-structure: An optimal solution to a problem must be a composition of optimal subproblem solutions, using a relatively simple combining operation.
- Repeated subproblems: The recursive algorithm solves a small number of distinct subproblems, but they are repeatedly solved many times.

This last property allows us to take advantage of memoization, store intermediate values, using the appropriate dictionary data structure, and reuse when needed.

Difference with greedy

- Greedy problems have the greedy choice property: locally optimal choices lead to globally optimal solution. We solve recursively one subproblem
- I.e. In DP we solve all possible subproblems, while in greedy we are bound for the initial choice

Difference with divide and conquer

- Both require recursive programming with subproblems with a similar structure to the original
- D & C breaks a problems into a small number of subproblems each of them with size a fraction of the original size (size/b).
- In DP, we break into many subproblems with smaller size, but often, their sizes are not a fraction of the initial size.

A first example: Fibonacci Recurrence.

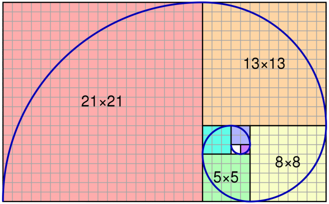
The Fibonacci numbers are defines recursively as follows:

$$
\begin{aligned}
F_0 &= 0\\
F_1 &= 1\\
F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n \geq 2
\end{aligned}
$$

0,1,1,2,3,5,8,13,21,34,55,89,..

↑

8th Fibonacci term

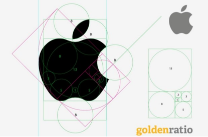


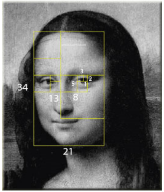
The golden ratio

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \varphi = 1.61803398875\ldots$$

Some examples of Fibonacci sequence in life

In nature, there are plenty of examples that follows a Fibonacci sequence pattern, from the shells of mollusks to the leaves of the palm. Below you have some further examples:





YouTube: Fibonacci numbers, golden ratio and nature

Computing the n-th Fibonacci number.

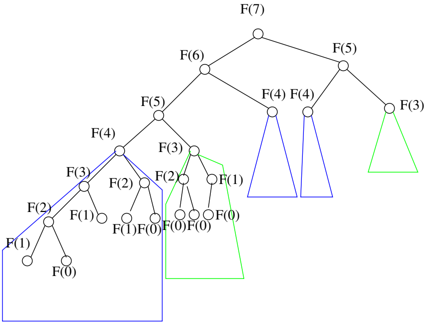
INPUT: $n \in \mathbb{N}$
QUESTION: Compute F_n.

A recursive solution:

```
Fibonacci ($n$)
if $n = 0$ then
    return 0
else if $n = 1$ then
    return 1
else
    return (Fibonacci($n - 1$)+Fibonacci($n - 2$))
```

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.



Notice the computation of subproblem $F(i)$ is repeated many times

A DP implementation: memoization

To avoid repeating multiple computations of subproblems, keep a dictionary with the solution of the solved subproblems.

```
**Fibo**($n$)
**for** $i \in [0..n]$ **do**
    $F[i] = -1$
$F[0] = 0$; $F[1] = 1$
**return** (**Fibonacci**($n$))
```

```
**Fibonacci** ($i$)
**if** $F[i] \neq -1$ **then**
    **return** ( $F[i]$)
$F[i]$= **Fibonacci**($i-1$) + **Fibonacci**($i-2$)
**return** ($F[i]$)
```

Each subproblem requires $O(1)$ operations, we have $n+1$ subproblems, so the cost is $O(n)$.

We are using $O(n)$ additional space.

A DP algorithm: tabulating

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 $i = 2$ to $n$ do
    $F[i] = F[i-1] + F[i-2]$
return ($F[n]$)
```

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.

A DP algorithm: reducing space

In the tabulating approach, we always access only the previous two values. We can reduce space by storing only the values that we will need in the next iteration.

```
**DP-Fibonacci** ($n$) {Construct table}
$p1 = 0$
$p2 = 1$
**for** $i = 2$ to $n$ **do**
    $p3 = p2 + p1$
    $p1 = p2$; $p2 = p3$
**return** (p3)
```

To get F_n need $O(n)$ time and $O(1)$ space.

Computing the n-th Fibonacci number: cost

INPUT: $n \in \mathbb{N}$
QUESTION: Compute F_n.

To get F_n the last algorithm needs $O(n)$ time and uses $O(1)$ space.

The initial recursive algorithm takes $O(1.6^n)$ time and uses $O(n)$ space

Do we have a polynomial time solution? NO the size of the input is $\log n$.
We use the term pseudopolynomial for algorithms whose running time is polynomial in the value of some numbers in the input.

Guideline to implement Dynamic Programming

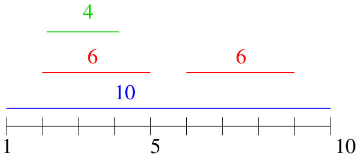
This first example of PD was easy, as the recurrence is given in the statement of the problem.

1. *Characterize the structure of subproblems:* 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, memoization/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:* compute additional information to trace-back from optimal solution from optimal value.

WEIGHTED ACTIVITY SELECTION problem

WEIGHTED ACTIVITY SELECTION problem: Given 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. Find the set of mutually compatible activities such that it maximizes $\sum_{i \in S} w_i$

Recall: We saw that some greedy strategies did not provide always a solution to this problem.



W Activity Selection: looking for a recursive solution

- Let us think of a backtracking algorithm for the problem.
- The solution is a selection of activities, i.e., a subset $S \subseteq \{1, \ldots, n\}$.
- We can adapt the backtracking algorithm to compute all subsets.
- When processing element i, we branch
    - i is in the solution S, then all activities that overlap with i cannot be in S.
    - i is not in S.

W Activity Selection: looking for a recursive solution

This suggest to keep at each backtracking call a partial solution (S) and a candidate set (C), those activities that are compatible with the ones in S.

```
WAS-1 ($S, C$)
if $C = \emptyset$ then
   return ($W(S)$)
Let $i$ be an element in $C$; $C = C - \{i\}$;
Let $A$ be the set of activities in $C$ that overlap with $i$
return ($\max\{$WAS-1$(S \cup \{i\}, C - A)$, WAS-1$(S, C)\}$)
```

The recursion tree have branching 2 and height $\leq n$, so size is $O(2^n)$.
How many subproblems appear here? hard to count better than $O(2^n)$.

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \cdots \leq f_n$.

```
WAS-2 $(S, i)$
if $i == 1$ then
    return $(W(S) + w_1)$
if $i == 0$ then
    return $(W(S))$
Let $j$ be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{\textbf{WAS-2}(S \cup \{i\}, j), \textbf{WAS-2}(S, i-1)\})$
```

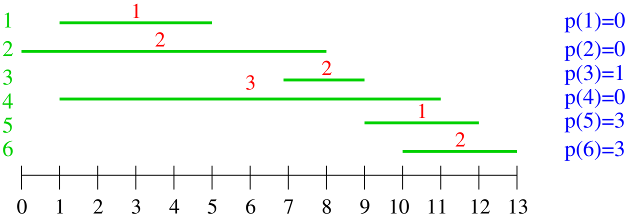
WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why? activities $j < k < i$ overlap with i any other that overlap with i also overlaps with j.

The algorithm has cost $O(2^n)$.

DP from WAS-2: a recurrence

- We need a $O(n \lg n)$ time for sorting.
- We have n activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights w_i, $1 \leq i \leq n$.
- Supproblems calls WAS-2(S, i)
    - S keeps track of the value of the solution
    - i defines de supproblem: W activity selection for activities $\{1, \ldots, i\}$, for $0 \leq i \leq n$.
    - $O(n)$ subproblems!
- Define $p(i)$ to be the largest integer $j < i$ such that i and j are disjoints ($p(i) = 0$ if no disjoint $j < i$ exists).
- Let $\mathrm{Opt}(j)$ be the value of an optimal solution O_j to the sub problem consisting of activities in the range 1 to j.

DP from WAS-2: a recurrence



Let Opt(j) be the value of an optimal solution O_j to the subproblem consisting of activities in the range 1 to j. Reinterpreting WAS-2, we get

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

DP from WAS-2: a recurrence

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

Correctness: From the previous discussion, we have two cases:

1.- $j \in O_j$:

- As j is part of the solution, no jobs $\{p(j)+1, \dots, j-1\}$ are in O_j,
- $O_j - \{j\}$ must be an optimal solution for $\{1, \dots, p[j])\}$, otherwise then $O'_j = O_{p[j]} \cup \{j\}$ will be better (optimal substructure)

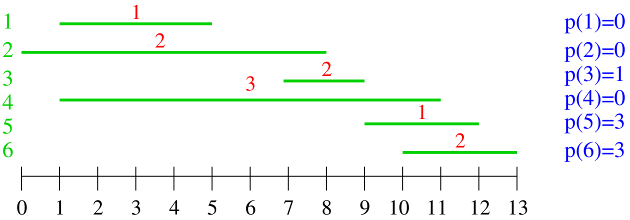
2.- If $j \notin O_j$: then O_j is an optimal solution to $\{1, \dots, j-1)\}$.

DP from WAS-2: Preprocessing

Considering the set of activities S, we start by a pre-processing phase:

- Sort the activities by increasing values of finish times.
- To compute the values of $p[i]$,
  - sort the activities by increasing values of start time.
  - merging the sorted list of finishing times an the sorted list of start times, in case of tie put before the finish times.
  - $p[j]$ is the activity whose finish time precedes s_j in the combined order, activity 0, if no finish time precedes s_j
- We can thus compute the p values in $O(n \lg n + n) = O(n \lg n)$

DP from WAS-2: Preprocessing



Sorted finish times: 1:5, 2:8, 3:9, 4:11, 5:12, 6:13

Sorted start times: 2:0, 1:1, 4:1, 3:7, 5:9, 6:10

Merged sequence: 2:0, 1:1, 4:1,1:5,3:7,3:9,5:9,6:10, 5:12, 6:13

DP from WAS-2: Memoization

We assume that tasks are sorted and all $p(j)$ are computed and tabulated in $P[1 \cdots n]$

We keep a table $W[n+1]$, at the end $W[i]$ will hold the weight of an optimal solution for subproblem $\{1, \dots, i\}$. Initially, set all entries to -1 and $W[0] = 0$.

```
R-Opt $(j)$
if $W[j]! = -1$ then
    return $(W[j])$
else
    $W[j] = \max(w_j + \textbf{R-Opt}(P[j])), \textbf{R-Opt}(j-1))$
    return $W[j]$
```

No subproblem is solved more than once, so cost is $O(n \log + n) = O(n \log n)$

DP from WAS-2: Iterative

We assume that tasks are sorted and all $p(j)$ are computed and tabulated in $P[1 \cdots n]$

We keep a table $W[n+1]$, at the end $W[i]$ will hold the weight of an optimal solution for subproblem $\{1, \ldots, i\}$.

```
Opt-Val ($n$)
$W[0] = 0$
for $j = 1$ to $n$ do
    $W[j] = \max(W[P[j]] + w_j, W[j-1])$
return $W[n]$
```

Time complexity: $O(n \lg n + n)$.

Notice: Both algorithms gave only the numerical max. weight

We have to keep more info to recover a solution form $W[n]$.

DP from WAS-2: Returning an optimal solution

To get also the list of activities in an optimal solution, we use W to recover the decision taken in computing $W[n]$.

```
Find-Opt ($j$)
if $j = 0$ then
    return $\emptyset$
else if $W[p[j]] + w_j > W[j-1]$ then
    return $(\{j\} \cup \text{Find-Opt}(p[j]))$
else
    return $(\text{Find-Opt}(j-1))$
```

Time complexity: $O(n)$

DP for Weighted Activity Selection

- We started from a suitable recursive algorithm, which runs $O(2^n)$ but solves only $O(n)$ different subproblemes.
- Perform some preprocesing.
- Compute the weight of an optimal solution to each of the $O(n)$ subproblems.
- Guided by optimal value, obtain an optimal solution .

0-1 KNAPSACK

(This example is from Section 6.4 in Dasgupta,Papadimritriou,Vazirani's book.)

0-1 KNAPSACK: Given as input a set of n items that can NOT be fractioned, item i has weight w_i and value v_i, and a maximum permissible weight W.

QUESTION: select a set of items S that maximize the profit.

Recall that we can NOT take fractions of items.

subproblems and and recurrence

Input: $(w_1, \dots, w_n)$, $(v_1, \dots, v_n)$, W.

- Let $S \subseteq \{1, \dots, n\}$ be an optimal solution to the problem The optimal benefit is $\sum_{i \in S} v_i$
- With respect to the last item we have two cases:
  - $n \notin S$, then S is an optimal solution to the problem $(w_1, \dots, w_{n-1})$, $(v_1, \dots, v_{n-1})$, W
  - $n \in S$, then $S - \{n\}$ is an optimal solution to the problem $(w_1, \dots, w_{n-1})$, $(v_1, \dots, v_{n-1})$, $W - w_n$
- in both cases we get an optimal solution of a subproblem in which the last item is removed and in which the maximum weight can be W or a value smaller than W.
- This identifies subproblems of the form $[i, x]$ that are knapsack instances in which the set of items is $\{1, \dots, i\}$ and the maximum weight that can hold the knapsack is x.

Subproblems and recurrence

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

To compute $v[i, x]$, the two possibilities we have considered give raise to the recurrence:

$$v[i, x] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max v[i-1, x-w_i] + v_i, v[i-1, x] & \text{otherwise} \end{cases}$$

DP algorithm: tabulating

Define a table $P[n+1, W+1]$ to hold optimal values for the corresponding subproblem.

```
Knapsack($i, x$)
for $i = 0$ to $n$ do
    $P[i, 0] = 0$
for $x = 1$ to $W$ do
    $P[0, x] = 0$
for $i = 1$ to $n$ do
    for $x = 1$ to $W$ do
        $P[i, x] = \max\{P[i-1, x], P[i-1, x - w[i]] + v[i]\}$
return $P[n, W]$
```

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

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

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

Recovering the solution

To compute the actual subset $S \subseteq I$ that is the solution, we modify the algorithm to compute also a Boolean table $K[n+1, W+1]$, so that $K[i, x]$ is 1 when the max is attained in the second alternative ($i \in S$), 0 otherwise.

```
Knapsack($i, x$)
for $i = 0$ to $n$ do
    $P[i, 0] = 0;\ K[i, 0] = 0$
for $x = 1$ to $W$ do
    $P[0, x] = 0;\ K[0, x] = 0$
for $i = 1$ to $n$ do
    for $x = 1$ to $W$ do
        if $P[i-1, x] \geq P[i-1, x-w[i]] + v[i]$ then
            $P[i, x] = P[i-1, x]$;
            $K[i, x] = 0$
        else
            $P[i, x] = P[i-1, x-w[i]] + v[i]$;
            $K[i, x] = 1$
return $P[n, W]$
```

Complexity: $O(nW)$

An example

	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

Recovering the solution

- To compute an optimal solution $S \subseteq I$, we use K to trace backwards the elements in the solution.
- $K[i, x]$ is 1 when the max is attained in the second alternative: $i \in S$.

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

Complexity: $O(nW)$

An 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	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

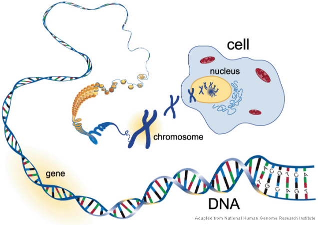
$K[5,11] \rightarrow K[4,11] \rightarrow K[3,5] \rightarrow K[2,0]$. So $S = \{4,3\}$

Complexity

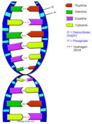
The 0-1 KNAPSACK is NP-complete.

- 0-1 KNAPSACK, has complexity $O(nW)$, and its length is $O(n \lg M)$ taking $M = \max\{W, \max_i w_i, \max_i v_i\}$.
- If W requires k bits, the cost and space of the algorithm is $n2^k$, exponential in the length W. However the DP algorithm works fine when $W = \Theta(n)$, here $k = O(\log n)$.
- Consider the unary knapsack problem, where all integers are coded in unary (7=1111111). In this case, the complexity of the DP algorithm is polynomial on the size, i.e., UNARY KNAPSACK $\in$P.

Matching DNA sequences



Adapted from National Human Genome Research Institute

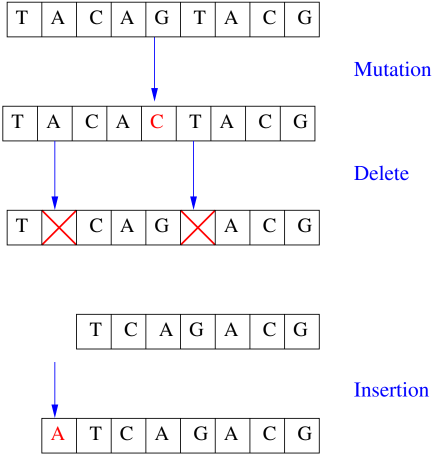


- DNA, is the hereditary material in almost all living organisms. They can reproduce by themselves.
- Its function is like a program unique to each individual organism that rules the working and evolution of the organism.
- Model as a string of 3×10^9 characters over $\{A, T, G, C\}$.

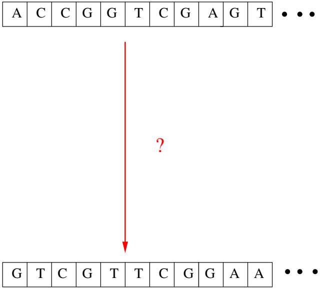
Computational genomics: Some questions

- 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 gene an those in 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



How to compare sequences?



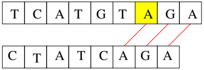
A C C G G T C G A G T • • •

?

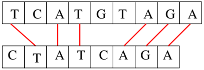
G T C G T T C G G A A • • •

Three problems

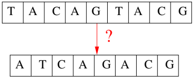
Longest common substring: Substring = consecutive characters in the string.



Longest common subsequence: Subsequence = ordered chain of characters (might have gaps).

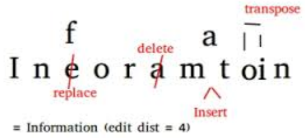


Edit distance: Convert one string into another one using a given set of operations.



The EDIT DISTANCE problem

(Section 6.3 in Dasgupta, Papadimritriou, Vazirani's book.)



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: Applications

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

EDIT DISTANCE: Levenshtein distance

In the Levenshtein distance the set of operations are

- 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 modify is 2, and the cost of insert/delete is 1.

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

For other operations and costs the structure of the DP will be similar.

Exemple-1

$X = aabab$ and $Y = babb$
$aabab = X$
$X' =$insert$(X, 0, b)$ $baabab$
$X'' =$delete$(X', 2)$ $babab$
$X'' =$delete$(X'', 4)$ $babb$
$X = aabab \to Y = babb$

A shortest edit distance

$aabab = X$
$X' =$modify$(X, 1, b)$ $babab$
$Y =$delete$(X', 4)$ $babb$

Use dynamic programming.

The structure of an optimal solution

- In a solution O with minimum edit distance from $X = x_1 \cdots x_n$ to $Y = y_1 \cdots y_m$, we have three possible alignments for the last terms

(1)	(2)	(3)
x_n	$-$	x_n
$-$	y_m	y_m

- In (1), O performs delete x_n, and it transforms optimally, $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.
- In (2), O performs insert y_m at the end of x, and it transforms optimally, $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- In (3), if $x_n \neq y_m$, O performs modify x_n by y_m, otherwise O, aligns them without cost. Furthermore O transforms optimally $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

The recurrence

Let $X[i] = x_1 \cdots x_i$, $Y[j] = y_1 \cdots y_j$.
$E[i, j]$ = edit distance from $X[i]$ to $Y[j]$ is the maximum of

- I *put y_j at the end of x*: $E[i, j-1] + 1$
- D *delete x_i*: $E[i-1, j] + 1$
- if $x_i \neq y_j$, M *change x_i into y_j*: $E[i-1, j-1] + 1$, otherwise $E[i-1, j-1]$

Edit distance: Recurrence

Adding the base cases, we have the recurrence

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

where

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

Computing the optimal costs and pointers

```
**Edit**$(X, Y)$
**for** $i = 0$ **to** $n$ **do**
    $E[i, 0] = i$
**for** $j = 0$ **to** $m$ **do**
    $E[0, j] = j$
**for** $i = 1$ **to** $n$ **do**
    **for** $j = 1$ **to** $m$ **do**
        $\delta = 0$
        **if** $x_i \neq y_j$ **then**
            $\delta = 1$
        $E[i, j] = E[i, j-1] + 1 \; b[i, j] = \uparrow$
        **if** $E[i-1, j-1] + \delta < E[i, j]$ **then**
            $E[i, j] = E[i-1, j-1] + \delta, \; b[i, j] := \nwarrow$
        **if** $E[i-1, j] + 1 < E[i, j]$ **then**
            $E[i, j] = E[i-1, j] + 1, \; b[i, j] := \leftarrow$
```

Space and time complexity: $O(nm)$.

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

Computing the optimal costs: 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	↖ 1	↖ 1	← 2	← 3
2	a	↑ 2	↖ 2	↖ 1	← 2	← 3
3	b	↑ 3	↖ 2	↑ 2	↖ 1	↖ 2
4	a	↑ 4	↑ 3	↖ 2	↑ 2	↖ 2
5	b	↑ 5	↖ 4	↑ 3	↑ 2	↖ 2

← is a I operation, ↑ is a D operation, and
↖ is either a M or a no-operation.

Obtain Y in edit distance from X

Uses as input the arrays E and b.

The first call to the algorithm is **con-Edit** (n, m)

```
con-Edit$(i, j)$
if $i = 0$ or $j = 0$ then
    return
    if $b[i, j] = \nwarrow$ and $x_i = y_j$ then
        change$(X, i, y_j))$; con-Edit$(i - 1, j - 1)$
    if $b[i, j] = \uparrow$ then
        delete$(X, i)$; con-Edit$(i - 1, j)$
    if $b[i, j] = \leftarrow$ then
        insert$(X, i, y_j)$, con-Edit$(i, j - 1)$
```

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

The Longest Common Subsequence

(Section 15.4 in CormenLRS' book.)

- $Z = z_1 \cdots z_k$ is a subsequence of X if there is a subsequence of integers $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ such that $z_j = x_{i_j}$.

  TTT is a subsequence of $ATATAT$.
- If Z is a subsequence of X and Y, then Z is a common subsequence of X and Y.

LCS Given sequences $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$, compute the longest common subsequence Z.

DP approach: Characterization of optimal solution

Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let Z be a longest common subsequence (lcs). Then,

- $Z = x_{i_1} \ldots x_{i_k} = y_{j_1} \ldots y_{j_k}$
- There are no i, j, with $i > i_k$ and $j > j_k$, s.t. $x_i = y_j$. Otherwise, Z will not be optimal.
- $a = x_{i_k}$ might appear after i_k in X, but not after j_k in Y, or viceversa.
- There is an optimal solution in which i_k and j_k are the last occurrence of a in X and Y respectively.

DP approach: Characterization of optimal solution

Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let $Z = x_{i_1} \ldots x_{i_k} = y_{j_1} \ldots y_{j_k}$ a lcs s.t. the index of the final common symbol in Z is its last occurrence in both X and Y.

Let $X^- = x_1 \cdots x_{n-1}$ and $Y^- = y_1 \cdots y_{m-1}$

- Let us look at x_n and y_m.
- If $x_n = y_m$, $i_k = n$ and $j_k = m$ so, $x_{i_1} \ldots x_{i_{k-1}}$ is a lcs of X^- and Y^-.

DP approach: Characterization of optimal solution

Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let $Z = x_{i_1} \ldots x_{i_k} = y_{j_1} \ldots y_{j_k}$ a lcs s.t. the index of the final common symbol in Z is its last occurrence in X and Y.

Let $X^- = x_1 \cdots x_{n-1}$ and $Y^- = y_1 \cdots y_{m-1}$

- Let us look at x_n and y_m.
- If $x_n \neq y_m$,
  - If $i_k < n$ and $j_k < m$, Z is a lcs of X^- and Y^-.
  - If $i_k = n$ and $j_k < m$, Z is a lcs of X and Y^-.
  - If $i_k <$ and $j_k = m$, Z is a lcs of X^- and Y.
  - The last two include the first one!

DP approach: Supproblems

Subproblems = lcs of pairs of prefixes of the initial strings.
Notation:

- $X[i] = x_1 \ldots x_i$, for $0 \leq i \leq n$
- $Y[j] = y_1 \ldots y_j$, for $0 \leq j \leq m$
- $c[i,j]$ = length of the LCS of $X[i]$ and $Y[j]$.
- Want $c[n,m]$ i.e. length of the LCS for X and Y.

DP approach: Recursion

Therefore, given X and Y

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

The recursive algorithm

```
$\textbf{LCS}(X, Y)$
$n = X.size(); \; m = Y.size()$
if $n = 0$ or $m = 0$ then
    return $0$
else if $x_n = y_m$ then
    return $1 + \textbf{LCS}(X^-, Y^-)$
else
    return $\max\{\textbf{LCS}(X, Y^-), \textbf{LCS}(X^-, Y)\}$
```

The algorithm makes 1 or 2 recursive calls and explores a tree of depth $O(n+m)$, therefore the time complexity is $2^{O(n+m)}$.

DP: tabulating

We need to find the correct traversal of the table holding the $c[i, j]$ values.

- Base case is $c[0, j] = 0$, for $0 \leq j \leq m$, and $c[i, 0] = 0$, for $0 \leq i \leq n$.
- To compute $c[i, j]$, we have to access

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

A row traversal provides a correct ordering.

- To being able to recover a solution we use a table b, to indicate which one of the three options provided the value $c[i, j]$.

Tabulating

```
**LCS**$(X, Y)$
for $i = 0$ to $n$ do
    $c[i, 0] = 0$
for $j = 1$ to $m$ do
    $c[0, j] = 0$
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
            $c[i, j] = c[i, j-1],\ b[i, j] = \uparrow$.
```

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

Following the arrows: TCTA

Construct the solution

Access the tables c and d.
The first call to the algorithm is **sol-LCS**(n, m)

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

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

Longest common substring

- A slightly different problem with a similar solution
- LCSt: Given two strings $X = x_1 \ldots x_n$ and $Y = y_1 \ldots y_m$, compute their longest common substring Z, i.e., the largest k for which there are indices i and j with $x_i x_{i+1} \ldots x_{i+k} = y_j y_{j+1} \ldots y_{j+k}$.
- For example:
  X : DEADBEEF
  Y : EATBEEF
  Z :

Longest common substring

- A slightly different problem with a similar solution
- LCSt Given two strings $X = x_1 \ldots x_n$ and $Y = y_1 \ldots y_m$, compute their longest common substring Z, i.e., corresponding to the largest k for which there are indices i and j with $x_i x_{i+1} \ldots x_{i+k} = y_j y_{j+1} \ldots y_{j+k}$.
- For example:
  X : DEADBBEEF
  Y : EATBEEF
  Z : BEEF pick the longest substring

Characterization of optimal solution

- Let $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ and let Z be a longest common substring.
    - $Z = x_i \ldots x_{i+k} = y_j \ldots y_{j+k}$
    - Z is the longest common suffix of $X(i + k)$ and $Y(j + k)$.
- We can consider the subproblems $LCStf(i, j)$: compute the longest common suffix of $X(i)$ and $Y(j)$.
- The $LCSf(X, Y)$ is the longest of such common suffixes.

Computing the LC Suffixes

- To solve $LCSf(i,j)$ it is enough to go backward from position i in X and j in Y until we find two different characters.
- This has cost $O(n+m)$ per subproblem.
- We get a $O(nm(n+m))$ algorithm for LCSt
- Can we do it faster? Let us use DP!

A recursive solution for LC Suffixes

Notation:

- $X[i] = x_1 \ldots x_i$, for $0 \leq i \leq n$
- $Y[j] = y_1 \ldots y_j$, for $0 \leq j \leq m$
- $s[i,j] =$ the length of the LC Suffix of $X[i]$ and $Y[j]$.
- Want $\max_{i,j} s[i,j]$ i.e., the length of the LCSt of X, Y.

DP approach: Recursion

Therefore, given X and Y

$$s[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 0 & \text{if } x_i \neq y_j \\ s[i-1,j-1]+1 & \text{if } x_i = y_j \end{cases}$$

Using the recurrence the cost per recursive call (or per element in the table) is constant

Tabulating

```
**LCSf**$(X, Y)$
for $i = 0$ to $n$ do
    $s[i, 0] = 0$
for $j = 1$ to $m$ do
    $s[0, j] = 0$
for $i = 1$ to $n$ do
    for $j = 1$ to $m$ do
        s[i,j]=0
        if $x_i = y_j$ then
            $s[i, j] = s[i-1, j-1] + 1$
```

complexity: $O(nm)$.

Which gives an algorithm with cost $O(nm)$ for LCSt