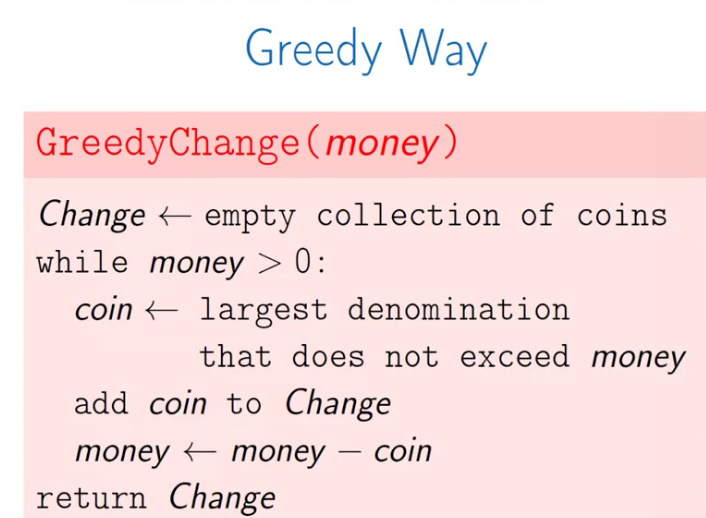
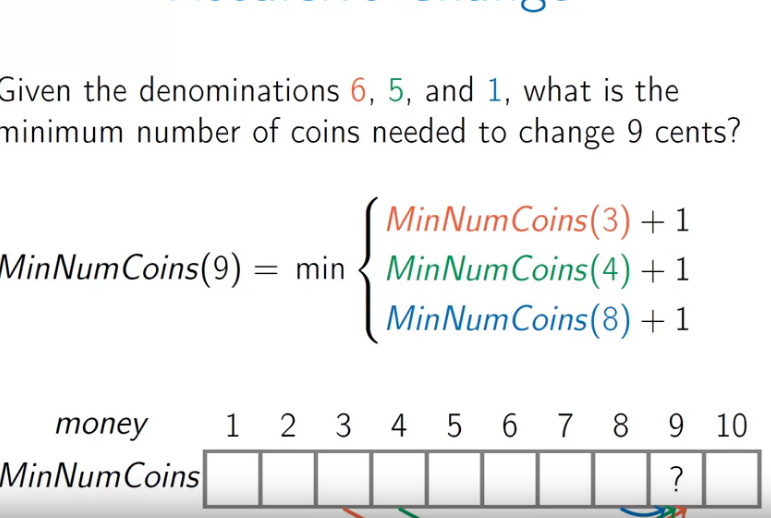
**Dynamic Programming**

* Revenim la problema in care trebuie sa aflam numarul minim de monede pentru a obtine o suma.
* Algoritmul pentru a face asta e asa:



* Totusi, acest algoritm care mereu cauta daca cea mai mare moneda e buna, nu merge
* De ex, daca avem 40 si avem monede de 25 20 10 5 si vrem sa avem 40 de monede cu bancnote minime, algoritmul ar returna 25 10 5, desi varianta optima e 20 20
* **Greedy alghoritm failed aici**
* Sa zicem ca vrem sa avem 9 monede din 6 5 si 1:



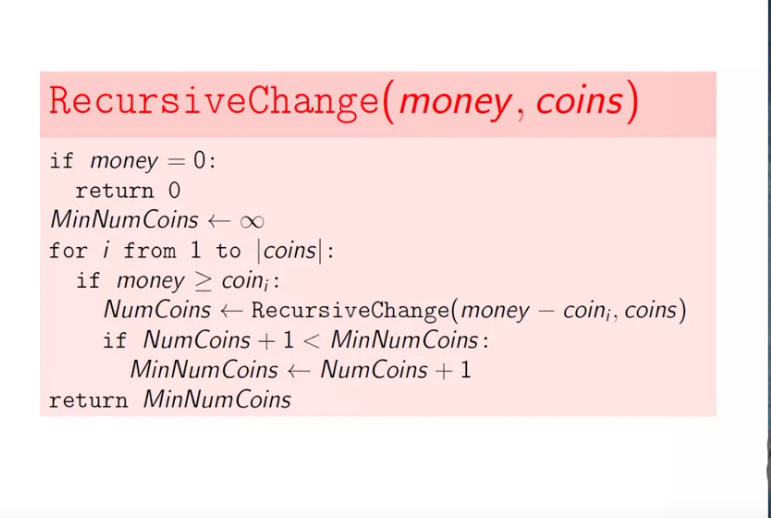
Avem 3 variante:

9 monede – 1 1 1 1 1 1 1 1 1

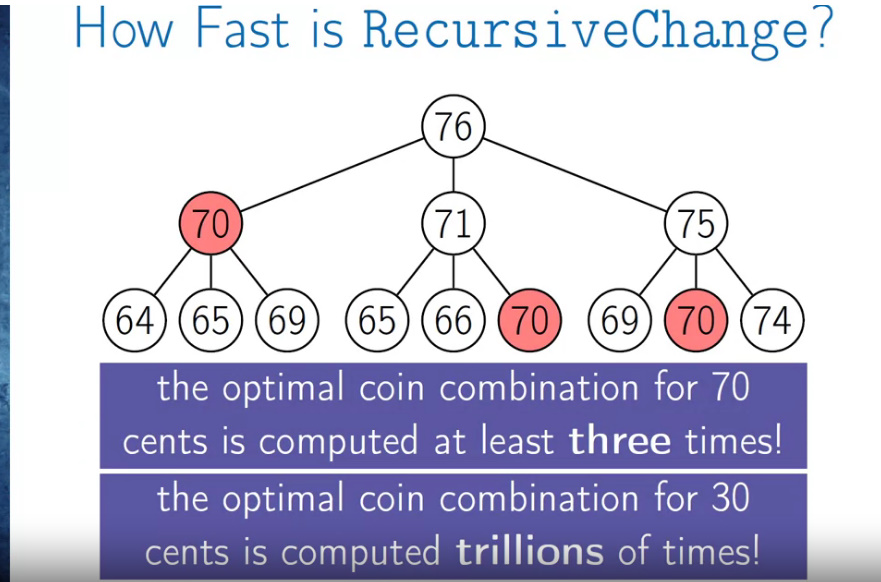
5 monede – 5 1 1 1 1

4 monede – 6 1 1 1

* Putem folosi recursivitatea:

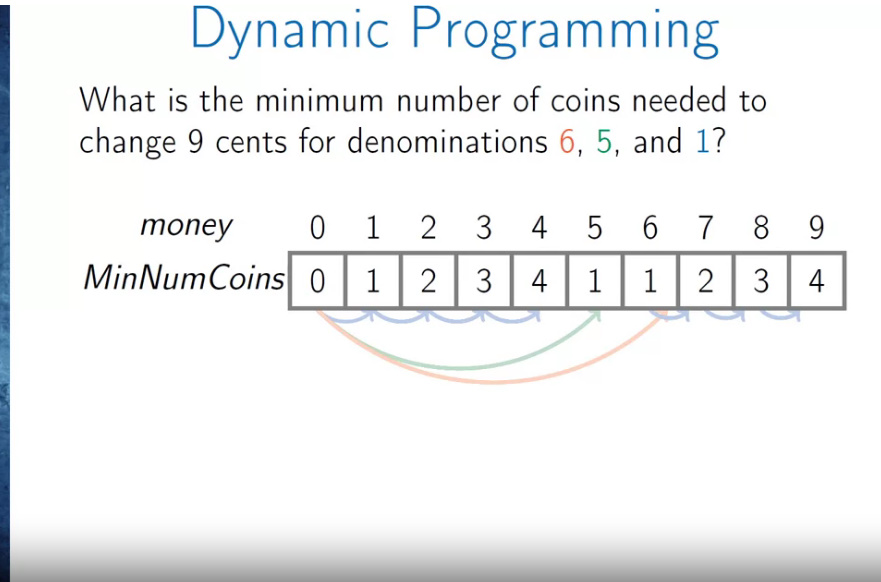


Acest algoritm e foarte lent si rau



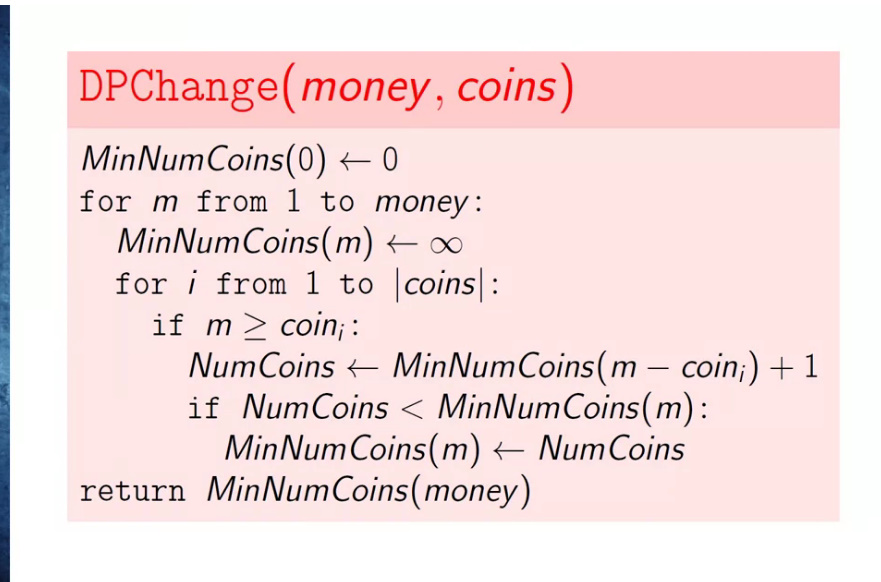
* Cu acest algoritm se obtin toate combinarile posibile de monede pentru a avea 76 in total si asta e imens.
* Putem in schimb folosi urmatoarea strategie de pogramare dinamica:

**Exemplu de programare dinamica**



In fiecare casuta, socotim de cate monede minim avem nevoie pentru a obtine 0, 1, 2,..9 monede din 6,5,1 pe baza la cele anterioare

* pentru 0 clar e 0
* pentru 1 avem o moneda de 1
* pentru 2 nu exista moneda, deci are nevoie de atatea monede cat a avut precedenta + 1
* pentru 3 nu exista, deci are nevoie de cate a avut precedenta + 1
* pentru 4 precedenta + 1
* pentru 5 exista o moneda, deci are nevoie de 1
* pentru 6 exista moneda, deci 1
* pentru 7 nu exista moneda, deci precedenta +1
* 8 si 9 precedenta + 1



Aici totusi putin facem altfel.

* primul if verifica daca pozitia, care corespundei si cifrei ce vrem sa o schimbam in monede, nu e mai mare sau egala ca moneda selectata, asa cum al 2 for va incerca toate monedele pentru pozitia aleasa
* daca pozitia e mai mica ca moneda, inseamna ca nicidecum moneda data nu poate fi folosita pentru suma data
* daca pozitia e mai mare sau egala ca moneda, inseamna ca avem la sigur o moneda, de acolo si +1 apare, si inca cateva monede de la elementul care se afla cu cateva pozitii in urma, pozitie egala cu pozitia elementului curent minus valoarea monedei.
* Apoi merge urmatoarea moneda si vede deja ea cate pozitii necesita.

La fiecare cifra ajunsa, vom verifica daca nu exista mai multe variante posibile. De asta se ocupa:

NumCoins <- MinNumCoins(m – coini) + 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 4 |

* pentru 2,3,4 se va executa if doar cand se va ajunge la moneda de 1, asa cum 2..4>=1

acum, 2 mai are nevoie de un numar de monede egal cu cel al termenului ce se afla cu un numar de pozitii in urma egal cu pozitia lui actuala minus cea a monedei + 1 moneda care la sigur merge, adica cea de 1:

2 = 1 + 1

3 are nevoie de o moneda la sigur, plus monedele de pe pozitia 3 – 1, adica 2,

3 = 1 + 2

....

* pentru 5, se va incerca varianta cu o moneda, deci

5 = 1 + monedele de la pozitia 5 – 1 = 1 + 4 = 5

5 = 1 + monedele de la pozitia 5 – 5 = 0, deci 1 + 0 = 0

* pentru 6:

6 = 1 + monedele de la poztia 6 – 6 = 1 + 0 = 1

6 = 1 + monedele de la pozitia 6 – 5 = 1 + 1 = 2, d**eci ia monede de care a avut nevoie termenul precedent**

6 = 1 + monedele de la pozitia 6 – 1 = 1 + 1 = 2

* pentru 9

9 = 1 + monede pozitia 9 - 1 = 1 + 4 = 5, alea de la 8 plus inca 1, adica cu moneda de 1 inca 8 de unu

9 = 1 + pozitia 9 – 5 = 1 + 4 = 5 , cu moneda de 5 inca 4

9 = 1 + pozitia 9 – 6 = 1 + 3 = 4, adica cu moneda de 6 inca 3 monede

* Programarea in termenul de Programare Dynamica nu are nimic de a face cu programarea propriu zisa!

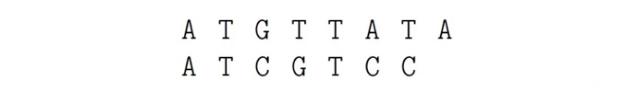
**Dynamic Programming**

* Programarea dinamica descompune o problema de optimizare in simple subprobleme si pastreaza solutia la fiecare subproblema astfel incat fiecare subproblema este rezolvata doar odata. Aceasta se uita la toate sub

problemele posibile fara a rezolva aceeasi subproblema din nou vreodata.

**Alignment game(Vezi document 2)**

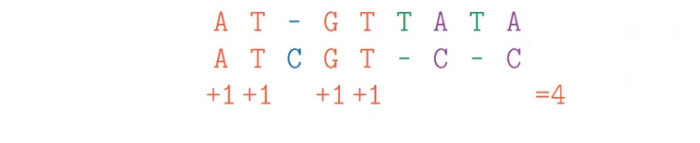
Sa zicem ca avem 2 linii de litere:



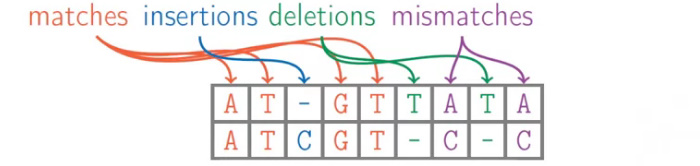
Daca comparam primul simbol din ambele stringuri si ele sunt egale, primim un punct, daca nu 0

Daca punem un space in loc de un caracter si impingem stringul, avem 0 puncte,

Scopul este de a obtine un scor de puncte maxim

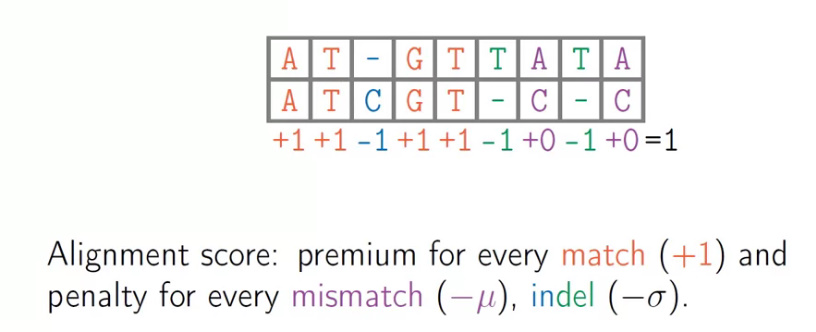


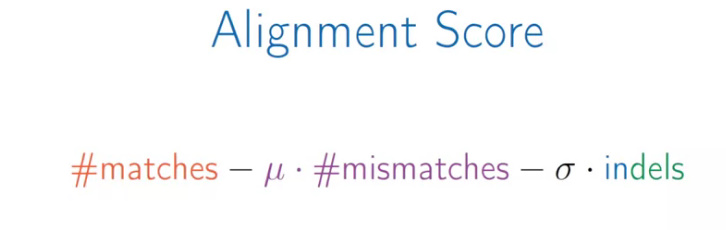
1. A/A ne da un punct
2. T/T 1 punct
3. Daca G ar fi deasupra lui C, am primi 0, dar dupa C merge un alt G,si deci in locul lui G de sus punem un loc liber,impingem subsirul inainte si avem G sub G
4. G/G = 1
5. T/T = 1



insertions se refera la faptul ca punem un space la linia de sus si impingem substringul inainte

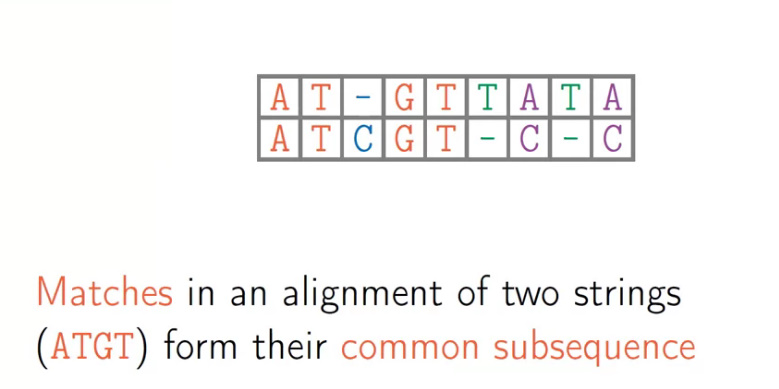
deletions – punem un space la linia de jos.

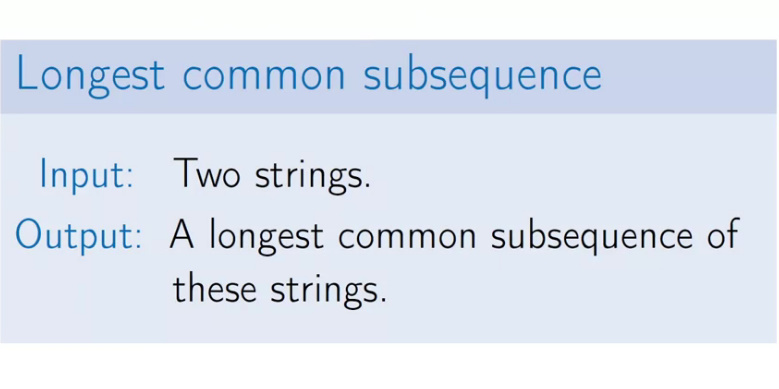
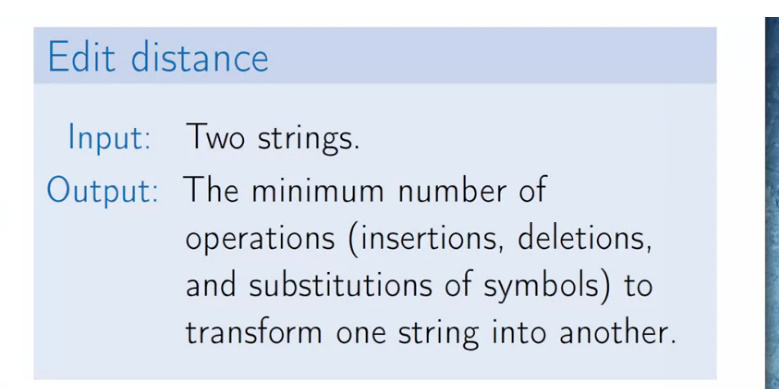


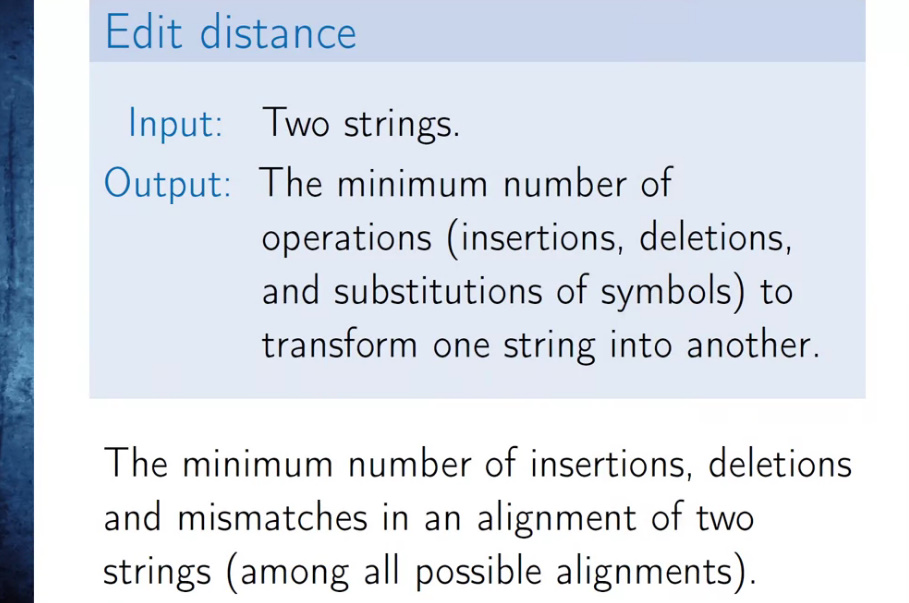


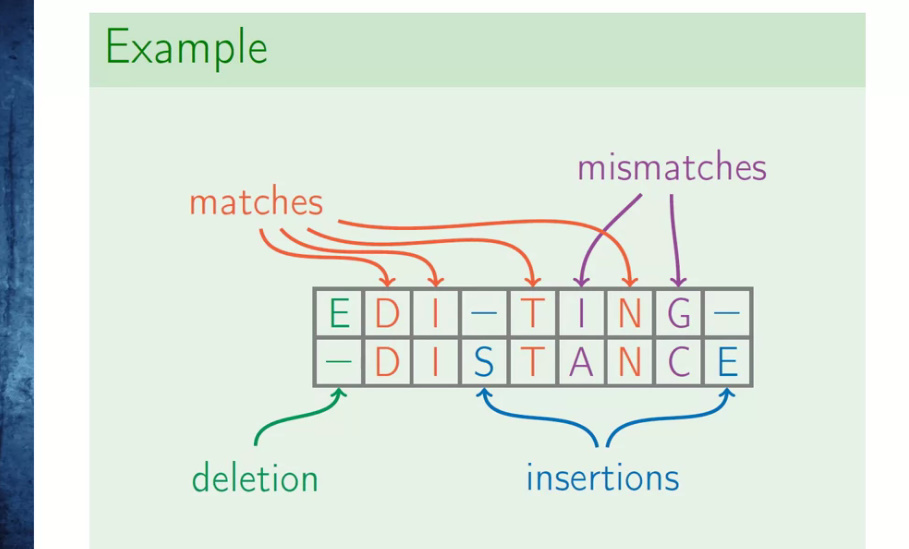
Noi putem stabili ce penalitati sa fie pentru mismatches si indels.

ATGT – stringul lor comun



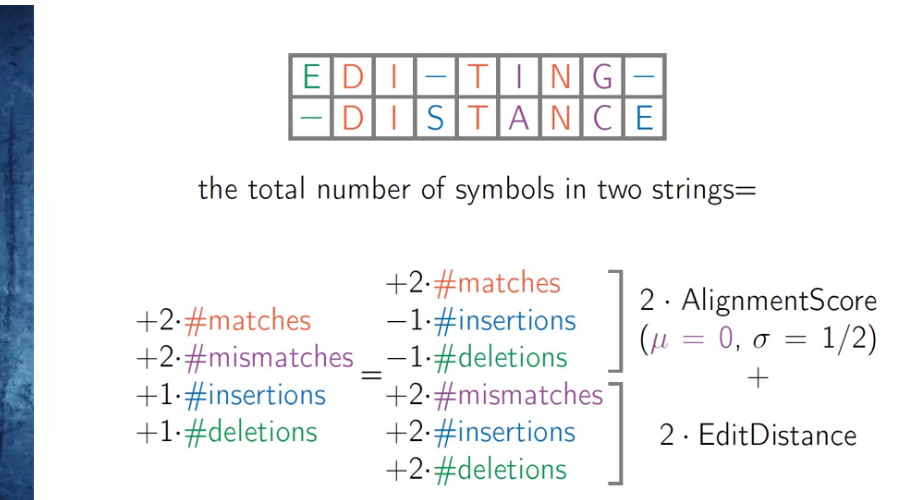






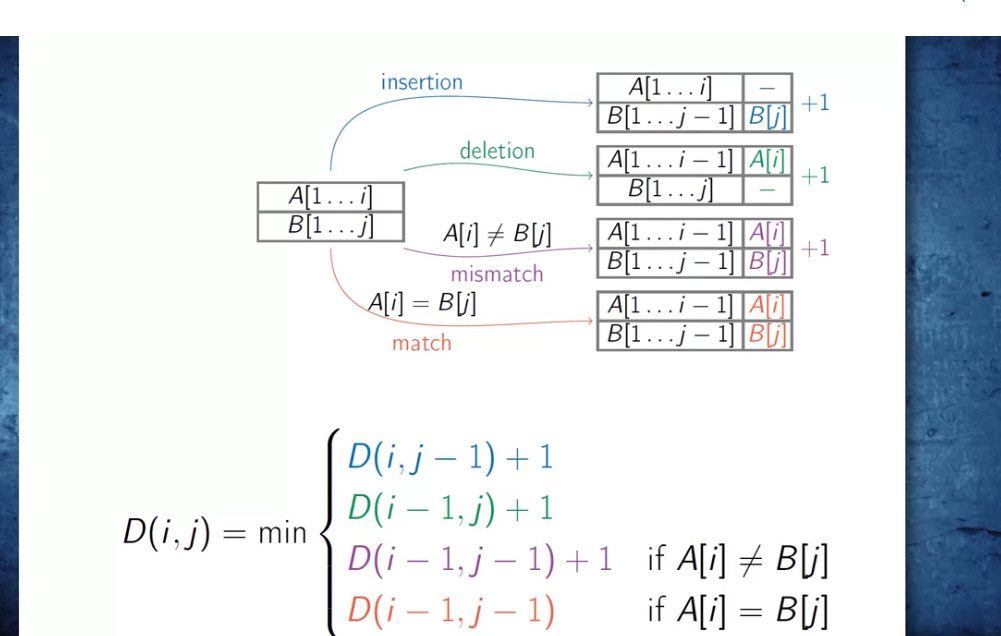
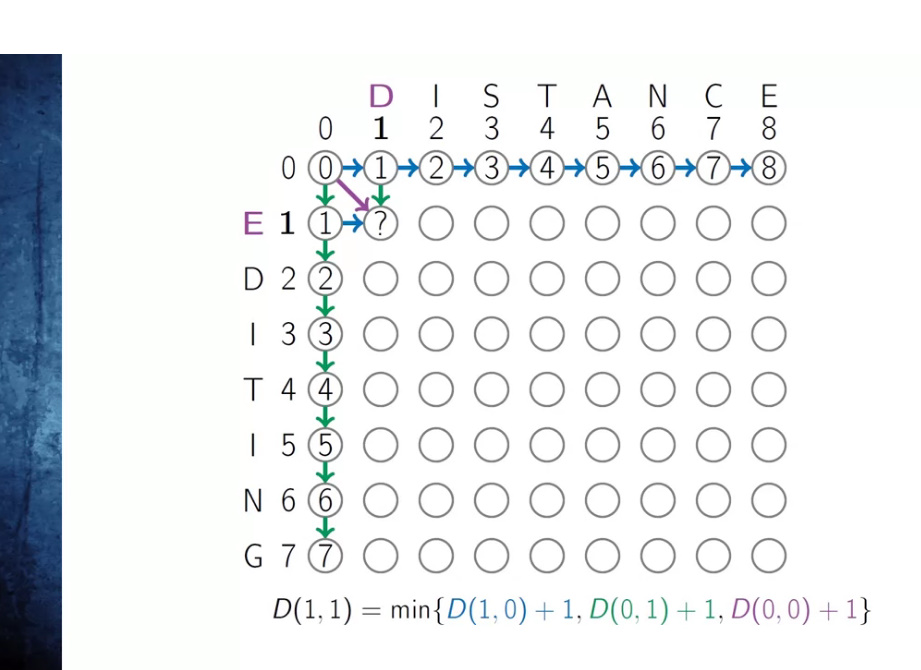
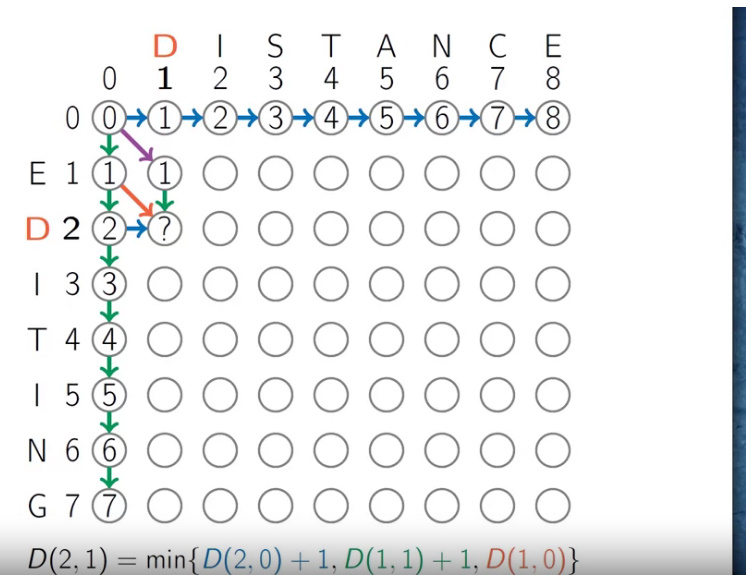
bread

really



* Din poza de mai sus, vedem ca numarul total de caractere in cele 2 stringuri e 2\*matches + 2\*mismatches + 1\*insertions + 1\*deletions
* Mai apoi doar rescriem putin, caci oricum 2\*insertions – 1\*insertions = 1\*insertions. Facem aceste transformari ca sa calculam un AlignmentscORE si EditDistance
* 2\*AlignmentScore = 2\*matches – 1\*insertions – 1\*deletions – adica calculam penalizarea totala
* ***Minimazing edit distance = maximazing alignment score***

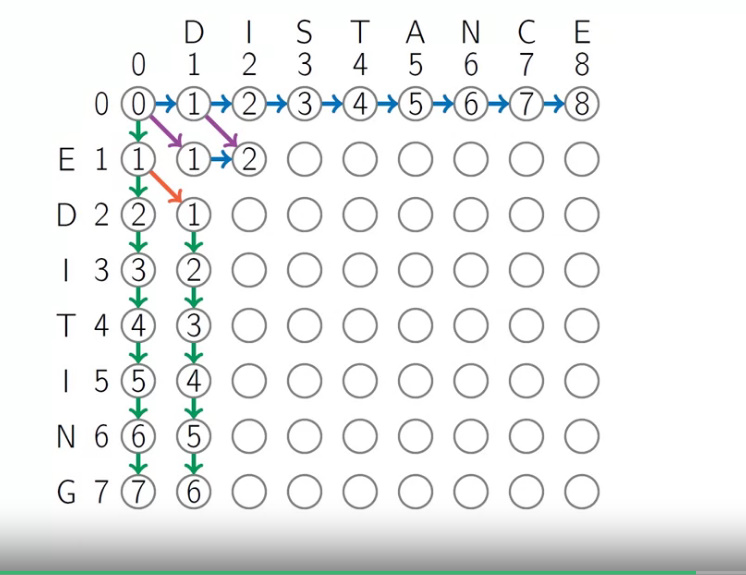
**Crearea Matricei**

* **Citeste fisierul al 2!!!**
* Putem sa completam matricea si putin altfel. Putem scrie in casete numarul de modificari ce trebuie de facut in sir pentru a obtine scorul cel mai mare.
* Edit Distance se refera la numarul de modificari
* 
* Deci, comparam toate drumurile posibile, si fiecare punct de la care pornim spre cel curent poate duce la aceea ca vom avea un isert, delete, compararea rea sau corecta. De aceea, aflam minimul din rezultatele obtinute din toate punctele din care venim, si care va rezulta cu o valoare mai mica de modificari, e cel mai bun.
* 
* Daca pornim de la 1 de langa D (0,1), vom avea inca o modificare in sir +1,deci 1 + 1 = 2, la fel si pentru 1 de langa E (1,0), insa daca pornim de la 0, vom avea inca o modificare +1 si deci inainte era de 0, acum doar de 1, Deci, punem 1, nr de modificari ce vine cand incepe de la 0.
* 

In poza coordonatele merg invers, nu de la axa x la y, ci de la y la x

Acum, putem sa incepem de la (2,0),(1,1) sau (1,0), adica de la 1,1 sau 2. Daca incepem de la (0,2), avem linie orizontala, deci D de sus va avea sub el un sptiu liber si avem penalizare 2 + 1 = 3. Daca de la (1,1) avem linie dreapta si deci D din stanga va avea desupra lui iar spatiu liber, deci 1 + 1modificari = 2.Acum avem (0,1) si va duce la comparare, si compararea e cu succes, deci avem 1 de la (0,1) ca unica modificare de pana acum. Asa dar, scriem in loc de ? 1 ca sa aratam ca avem pana acum doar o modificare

* Totusi, se poate intampla sa avem 2 valori optime, in asa caz le alegem pe ambele:

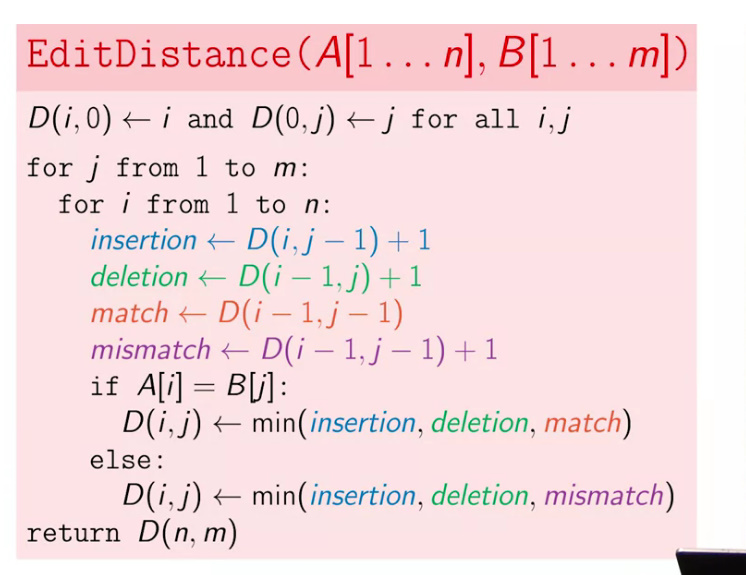


* Ultimul element din ultima linie e rezultatul final, asa cum putem avea si situatii de genul:

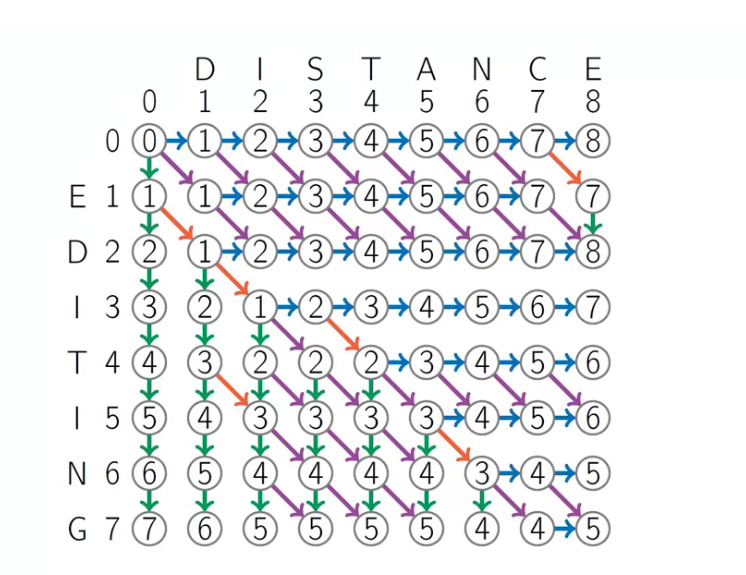
editing

distance

Acesta e doar ex, asa ceva nu e va intampla clar. Aici ideea e ca putem sa avem situatia cand se literele sunt puse asa ca stringul de sus nu mai are caarctere si cel de jos are, deci asta e ca si cum insertion, adica dupa editing mai sunt ca si cum 2 de - - sau spatii libere, si tot trebuie calculate.

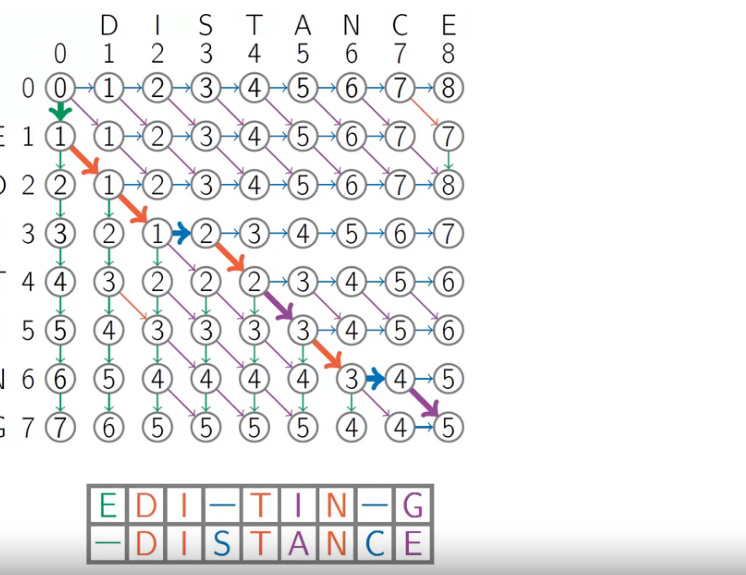


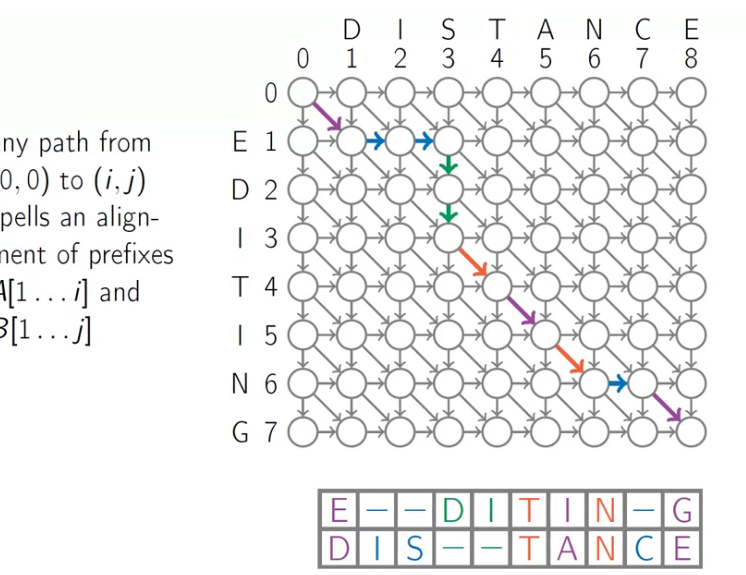
**Reconstruirea drumului**

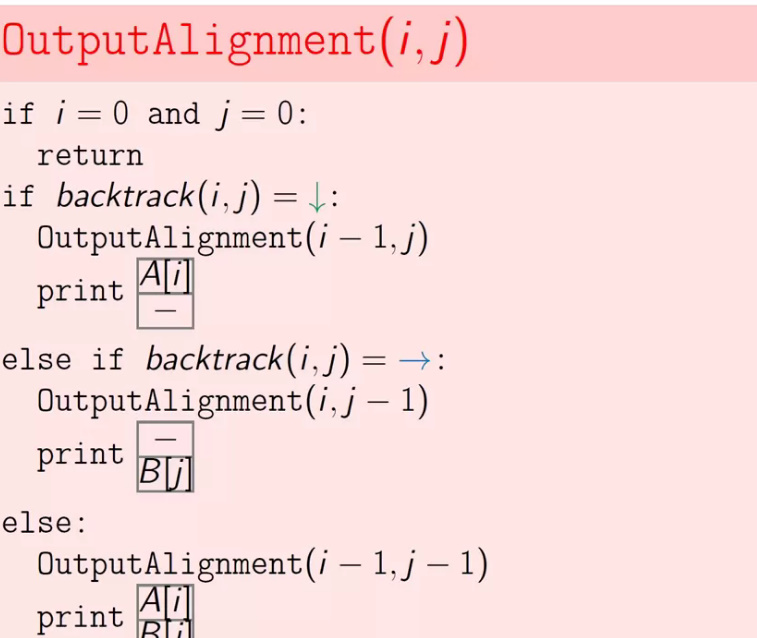


Ne bazam pe faptul ca linia orizontala inseamna ca se pun spatii dupa caracterul orizontal ce-i corespunde si cea verticala pune spatii dupa caracterului pe verticala ce corespunde. Mai sus avem un exemplu

* Pentru a reconstrui drumul, trebuie sa folosim pointerii. Fiecare element va trebui sa memoreze de la ce element a venit.







Putem sa mai economisim spatiu:

