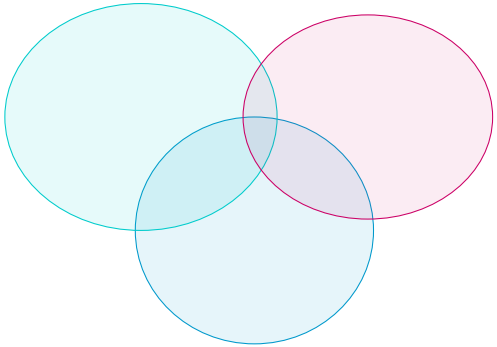
Algorithmique 1
Programmation dynamique

Hajer Akid & Sandrine Julia

Semestre 4

Des sous-problèmes s'intersectent ?...



Principe de la **programmation dynamique** :
Stocker chaque solution partielle pour ne **jamais** effectuer 2 fois le même calcul !

Coefficients binomiaux

- un **coefficient binomial** $\binom{n}{k}$ se définit par :

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- ce sont les coefficients obtenus dans la formule du **binôme de Newton** :

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k}\, a^k\, b^{n-k}$$

- la somme de $\binom{n}{k}$ pour k variant de 0 à n est donc égale à 2^n
- $\binom{n}{k}$ compte le **nombre de combinaisons** de k éléments pris parmi n **sans ordre** et **sans répétition**
- par convention, $\binom{n}{k} = 0$ pour tout $k > n$

Triangle de Pascal

Représentation des coefficients $\binom{n}{k}$ où n est le n° de ligne et k le n° de colonne :

1															
1	1														
1	2	1													
1	3	3	1												
1	4	6	4	1											
1	5	10	10	5	1										
1	6	15	20	15	6	1									
1	7	21	35	35	21	7	1								
1	8	28	56	70	56	28	8	1							
1	9	36	84	126	126	84	36	9	1						
1	10	45	120	210	252	210	120	45	10	1					
1	11	55	165	330	462	462	330	165	55	11	1				
1	12	66	220	495	792	924	792	495	220	66	12	1			
1	13	78	286	715	1287	1716	1716	1287	715	286	78	13	1		
1	14	91	364	1001	2002	3003	3432	3003	2002	1001	364	91	14	1	
1	15	105	455	1365	3003	5005	6435	6435	5005	3003	1365	455	105	15	1

Triangle de Pascal

```
1
1     1
1     2     1
1     3     3     1     k=4
1     4     6     4     1
1     5     10    10    5     1
1     6     15    20    15    6     1
1     7     21    35    35    21    7     1
1     8     28    56    70    56    28    8     1
1     9     36    84    126   126   84    36    9     1
1     10    45    120   210   252   210   120   45    10    1
1   n=11    55    165   330  + 462  462   330   165   55    11    1
1     12    66    220   495  = 792  924   792   495   220   66    12    1
1     13    78    286   715   1287  1716  1716  1287  715   286   78    13    1
1     14    91    364   1001  2002  3003  3432  3003  2002  1001  364   91    14    1
1     15    105   455   1365  3003  5005  6435  6435  5005  3003  1365  455   105   15    1
```

Propriété

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

Exemple

$$\binom{12}{5} = \binom{11}{4} + \binom{11}{5} = 330+462 = 792$$

Algorithme récursif

```
entier Binomial_rec(entier n, entier k) {
  si (k>n) {
    retourner 0
  }
  sinon si (k=0 ou k=n){
    retourner 1
  }
  sinon {
    retourner Binomial_rec(n-1,k-1) + Binomial_rec(n-1,k)
  }
}
```

Complexité : $O(\varphi^n)$ **avec** $\varphi = \frac{1+\sqrt{5}}{2} \simeq 1,618$

Attention : les appels récursifs n'étant pas indépendants, il y a des "recalculs" !

A défaut de pouvoir séparer les appels, la **mémoïsation** permet d'éviter de recalculer.

Trace des appels récursifs

Appel à Binomial_rec (5, 3) :

```
Binomial_rec (5, 3)
    Binomial_rec (4, 2)
        Binomial_rec (3, 1)
            Binomial_rec (2, 0)
            1
            Binomial_rec (2, 1)
                Binomial_rec (1, 0)
                1
                Binomial_rec (1, 1)
                1
            2
        3
        Binomial_rec (3, 2)
            Binomial_rec (2, 1)
                Binomial_rec (1, 0)
                1
                Binomial_rec (1, 1)
                1
            2
            Binomial_rec (2, 2)
            1
    3
6

⋮  ⋮  ⋮
```

```
⋮  ⋮  ⋮

    Binomial_rec (4, 3)
        Binomial_rec (3, 2)
            Binomial_rec (2, 1)
                Binomial_rec (1, 0)
                1
                Binomial_rec (1, 1)
                1
            2
            Binomial_rec (2, 2)
            1
        3
        Binomial_rec (3, 3)
        1
    4
10
```


Algorithme itératif (prog. dyn.)

```
entier Binomial(entier n, entier k) {
  table ← matriceNulle(n+1,n+1)
  pour (i←0,i≤n,i++) {
    m ← min(i,k)
    pour (j←0,j≤m,j++) {
      si (j=0 ou i=j) {
        table[i,j] ← 1
      }
      sinon {
        table[i,j] ← table[i-1,j-1] + table[i-1,j]
      }
    }
  }
  retourner table[n,k]
}
```

Complexité : $O(n^2)$ **?**

Complexité grands entiers (en $x = \log(n)$**) : exponentiel**

table Binomial(5,3) :

```
1
1     1
1     2     1
1     3     3     1
1     4     6     4
1     5     10    10
```

Algorithme itératif (prog. dyn.)

```
entier Binomial(entier n, entier k) {
  table ← matriceNulle(n+1,n+1)
  pour (i←0,i≤n,i++) {
    m ← min(i,k)
    pour (j←0,j≤m,j++) {
      si (j=0 ou i=j) {
        table[i,j] ← 1
      }
      sinon {
        table[i,j] ← table[i−1,j−1] + table[i−1,j]
      }
    }
  }
  retourner table[n,k]
}
```

Complexité : $O(n^2)$ **?**

Complexité grands entiers (en $x = \log(n)$**) : exponentiel**

Attention !

se méfier des complexités d'apparence polynomiale mais qui doivent être calculées sur des grands entiers ... et qui en deviennent exponentielles.

Sous-mot (ou sous-chaîne)

- un **sous-mot** ou une sous-suite de lettres de S_1 est un mot obtenu à partir de S_1 en effaçant des lettres (pas forcément consécutives)
- un **sous-mot commun** ou sous-suite commune de deux mots est un sous-mot commun à ces deux mots
- on s'intéresse **aux sous-mots communs de longueur maximale**

Exemple

Soient les mots $S_1 = TAGTCACG$ et $S_2 = AGACTGT$ sur l'alphabet $\{A, C, G, T\}$:

- les chaînes AGC, $TATC$ et $AACG$ sont des sous-mots de S_1
- les chaînes GG, AGC et $AGTG$ sont des sous-mots communs à S_1 et S_2
- quelle est la plus longue sous-chaîne commune à S_1 et S_2 ?

Problème LCS (*Longest Common Subsequence Problem*)

Problème de la plus longue sous-chaîne

Données

- deux chaînes de caractères S_1 et S_2

Problème

- trouver une plus longue sous-chaîne commune à S_1 et à S_2

Exemple

- la chaîne $S_1 = TAGTCACG$
- la chaîne $S_2 = AGACTGT$
- une plus longue sous-chaîne commune à S_1 et S_2 est $AGACG$

Effectivement, $S_1 = TAGTCACG$ et $S_2 = AGACTGT$ et il n'en existe pas de longueur 6.

Scénario d'un algorithme

L'indice i parcourt la chaîne S, l'indice j la chaîne T :

Pour un algorithme récursif :

- l'élément *fil-rouge* est le dernier élément
- on compare les 2 lettres courantes $S[i]$ et $T[j]$
  - s'il y a concordance :
    - on ajoute cette lettre au résultat
    - on lance un appel récursif sur les 2 chaînes privées de cette lettre commune
  - sinon, on lance un appel récursif pour chaque possibilité :
    - S inchangée et T privée de sa dernière lettre
    - S privée de sa dernière lettre et T inchangée
- on combine les solutions des sous-problèmes renvoyées par les appels récursifs pour obtenir une solution

Algorithme récursif

```
entier plusLongue(chaine S, chaine T) {
  si (longueur(S) < longueur(T)) {
    retourner T
  }
  retourner S
}
```

```
chaîne LCS (chaine S, chaine T, entier i, entier j){
  // on considère les préfixes S[1..i] et T[1..j]
  si (i=0 ou j=0) {
    retourner
  }
  si (S[i] = T[j]) {
    retourner LCS(S,T,i-1,j-1) + S[i]
  }
  retourner plusLongue(LCS(S,T,i,j-1),LCS(S,T,i-1,j))
}
```

Complexité : $O(2^n), n = |S| + |T|$

Clairement, les "recalculs" ne sont pas évités ...

Algorithme itératif (prog. dyn.)

```
chaîne LCS (chaine S, chaine T, entier i, entier j){
  n ← longueur(S)
  m ← longueur(T)
  table ← matriceNulle(0..n,0..m)
  pour (i←1,i≤n,i++) {
    pour (j←1,j≤m,j++) {
      si (S[i] = T[j]) {
        a ← table[i-1,j-1]
        table[i,j] ← a + S[i]
      }
      sinon {
        b ← table[i,j-1]
        c ← table[i-1,j]
        table[i,j] ← plusLongue(b,c)
      }
    }
  }
  retourner table[n,m]
}
```

Complexité : $O(n.m)$

Complexité en espace : $O(n.m)$

Toutes les solutions partielles ont été sauvegardées !

Trace de l'algorithme itératif

Plus longue chaîne commune entre $S_1 = TAGTCACG$ et $S_2 = AGACTGT$?

		A	G	A	C	T	G	T
	ε	ε	ε	ε	ε	ε	ε	ε
T	ε	ε	ε	ε	ε	T	T	T
A	ε	A	A	A	A	A	A	A
G	ε	A	AG	AG	AG	AG	AG	AG
T	ε	A	AG	AG	AG	AGT	AGT	AGT
C	ε	A	AG	AG	AGC	AGC	AGC	AGC
A	ε	A	AG	AGA	AGA	AGA	AGA	AGA
C	ε	A	AG	AGA	AGAC	AGAC	AGAC	AGAC
G	ε	A	AG	AGA	AGAC	AGAC	AGACG	**AGACG**

Applications du problème LCS

Voici quelques applications directes du problème de la plus longue sous-chaîne commune :

- la **comparaison de fichiers** (`diff` d'UNIX)
- la **distance d'édition**
- la **reconnaissance de la parole**
- la détection de **similarités** (dans l'ADN ou autre)

La mémoïsation

Qu'est-ce ?

- une technique pour étendre l'idée de la programmation dynamique aux algoritmes récursifs
- les calculs effectués sont stockés au fur et à mesure dans un *dictionnaire* ou une *table de hachage*
- avant de calculer la valeur d'une fonction lors d'un appel récursif, on recherche si cette valeur est disponible
- ainsi, un calcul est effectué au plus une fois

Exemple

On aurait pu notablement abaisser la complexité de Binomial_rec(n,k) en utilisant la mémoïsation.