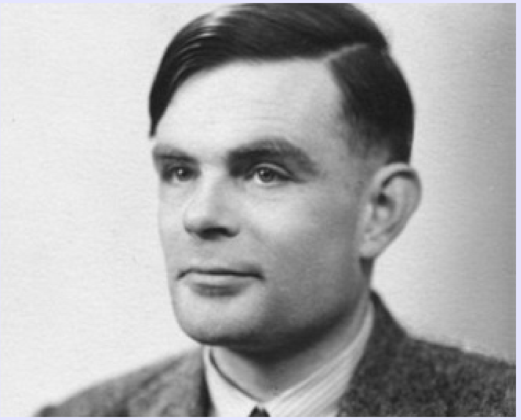

Ada Byron


Alan Turing


Grace Murray

Information, Calcul et Communication

Module 1 : Calcul

Leçon I.4 : Concevoir un algorithme

J.-C. Chappelier & J. Sam

Objectifs de la leçon

Dans la leçon précédente, nous avons vu comment évaluer l'ordre de **complexité** d'un algorithme en nous familiarisant avec deux familles de problèmes classiques: la recherche d'un élément dans un ensemble et le tri.

L'exemple du **tri par insertion** nous a permis d'illustrer **l'approche descendante** de **conception** d'un algorithme, d'abord en décomposant en une suite (claire) de sous-problèmes que nous avons ensuite traduits en séquences d'instructions.

Cette semaine nous allons approfondir la stratégie **récursive**, déjà illustrée avec l'exemple de **la recherche par dichotomie.** L'exemple de mise en œuvre de la récursivité avec le **tri fusion** (merge sort) illustrera aussi de manière plus complète la stratégie «diviser pour règner».

Enfin une troisième stratégie de conception, appelée la **programmation dynamique,** consiste à mémoriser des résultats intermédiaires pour améliorer l'ordre de complexité d'un problème. Elle sera illustrée sur un exemple de problème de recherche de chemin.

Plan

- Stratégie de conception récursive
  - Les tours de Hanoï
  - Somme des n premiers entiers
- Stratégie de conception «Diviser pour règner»
  - Le tri fusion
- Stratégie de conception par programmation dynamique
  - Coefficient binomial
  - Le plus court chemin

Récursion

Le principe de l'approche récursive est de

> *ramener le problème à résoudre à un sous-problème*, qui est une version simplifiée *du problème d'origine*.

Exemples :

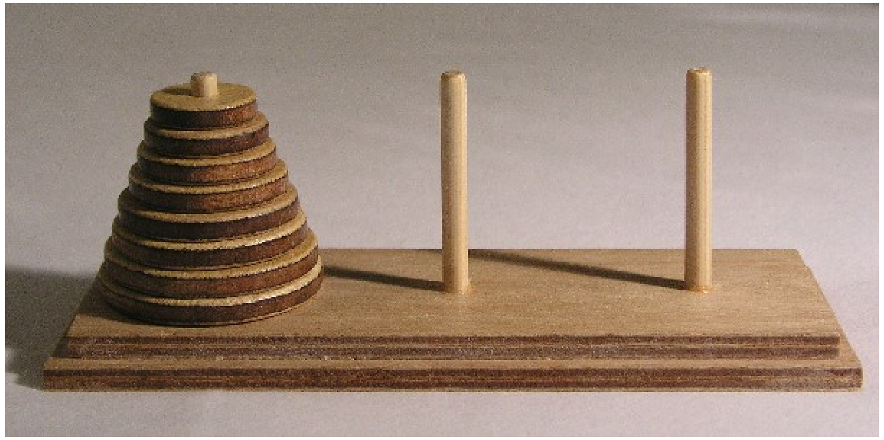
- recherche par dichotomie (cf leçon précédente)
  - La recherche récursive s'effectue *sur un ensemble plus petit*
- en mathématiques : le raisonnement par récurrence

Exemple : Les tours de Hanoï

But du jeu: Déplacer une colonne de disques de tailles décroissantes, d'un pilier de départ à un pilier d'arrivée

Règles du jeu:

- en utilisant un seul pilier de transition (il n'y a que 3 piliers en tout)
- en ne déplaçant qu'un seul disque à chaque fois
- en ne posant un disque que sur le sol ou sur un disque plus grand.

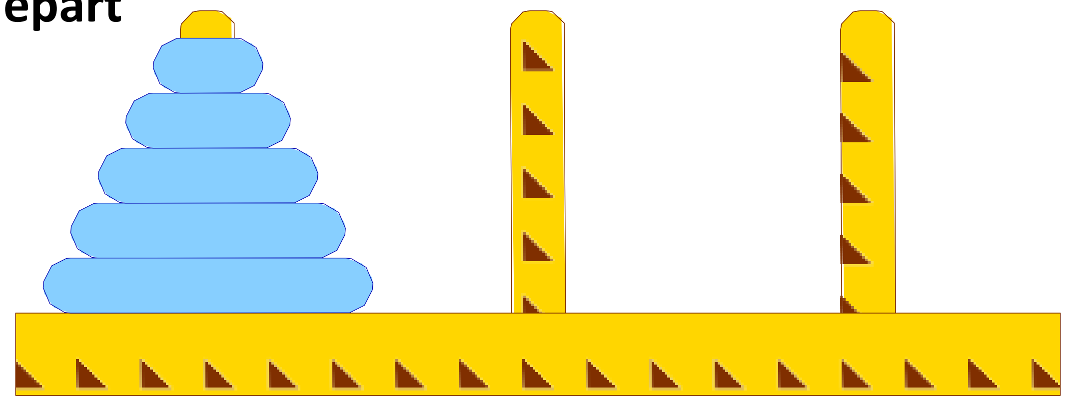


User:Evanherk (Wikimedia Commons)

Les tours de Hanoï (2)

Idée : si je peux le faire pour une pile de ***n*** disques,
je peux le faire pour une pile de ***n*** **+ 1** disques
(et je sais le faire pour une pile de **1** disque)

Démonstration du transfert de la pile de n+1 disques du pilier A au pilier C

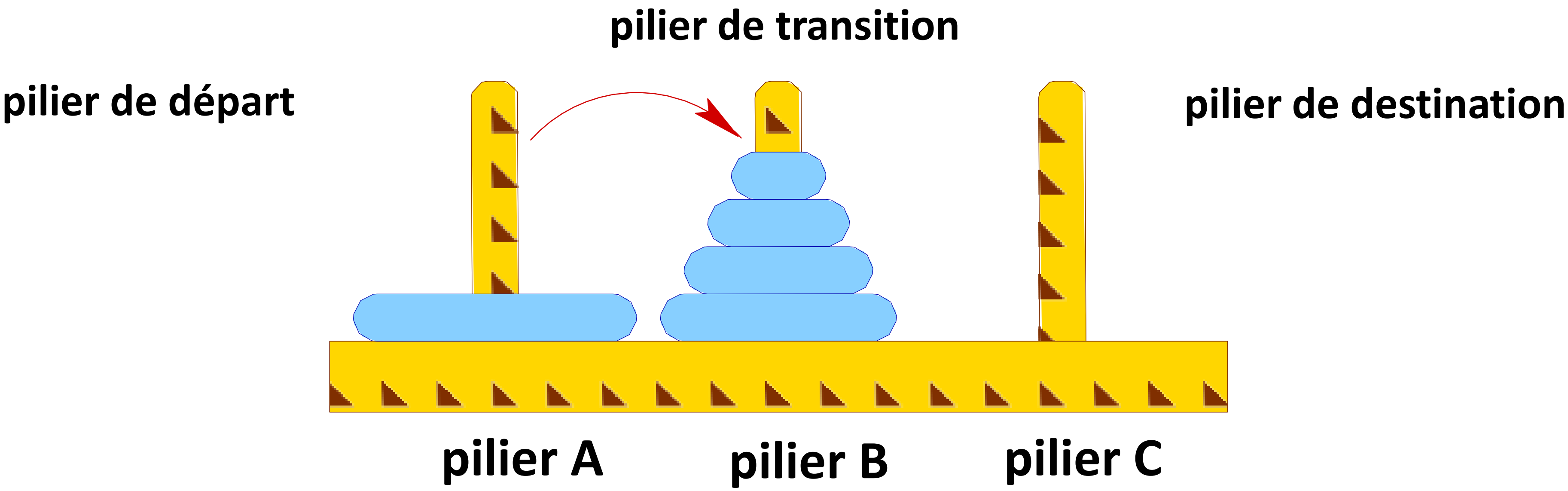


Les tours de Hanoï (2)

Idée : si je peux le faire pour une pile de ***n*** disques,
je peux le faire pour une pile de ***n*** **+ 1** disques
(et je sais le faire pour une pile de 1 disque)

Démonstration :

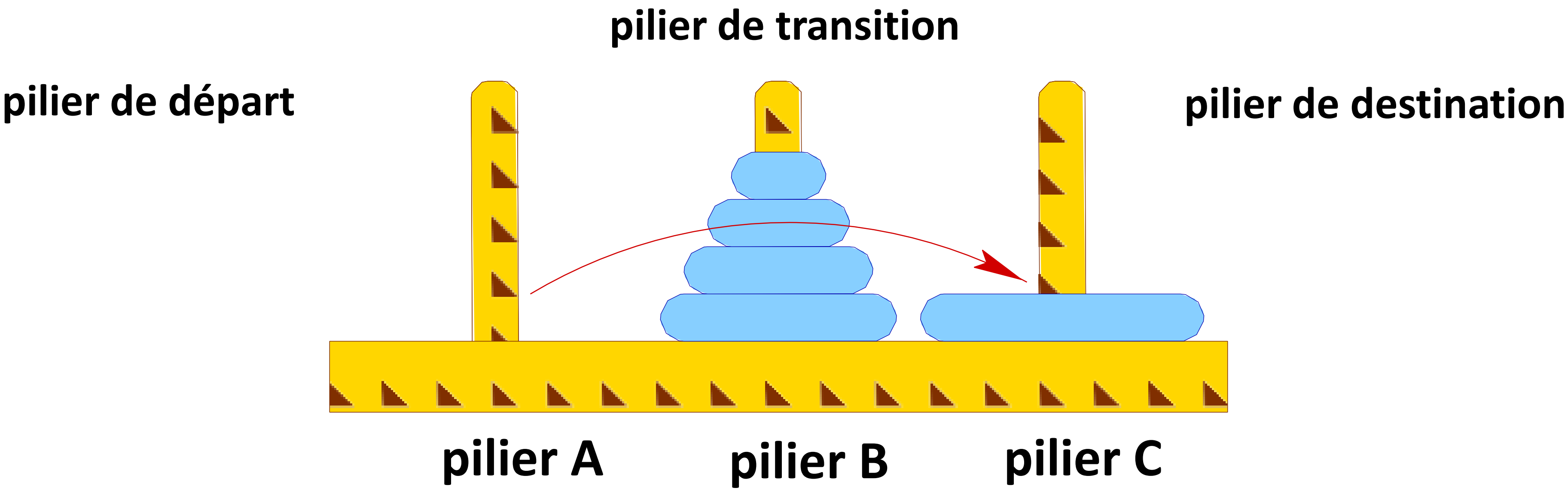
- je déplace les ***n*** disques du haut sur le pilier de transition
  (en utilisant la méthode que je connais par hypothèse)



Les tours de Hanoï (2)

Démonstration :

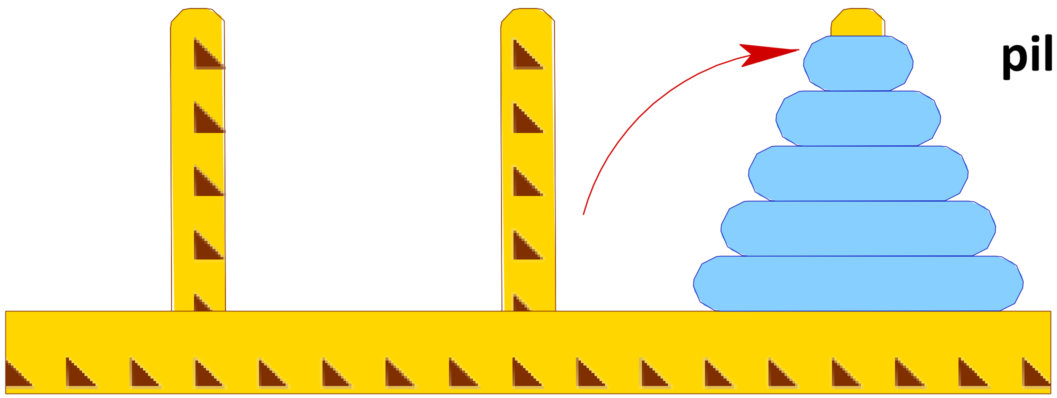
- je déplace les *n* disques du haut sur le pilier de transition (en utilisant la méthode que je connais par hypothèse)
- je mets le dernier disque sur le pilier destination



Les tours de Hanoï (2)

Démonstration :

- je déplace les *n* disques du haut sur le pilier de transition (en utilisant la méthode que je connais par hypothèse)
- je mets le dernier disque sur le pilier destination
- je redéplace la tour de *n* disques du pilier de transition au pilier destination (en utilisant à nouveau la méthode que je connais par hypothèse, et le pilier initial comme transition).

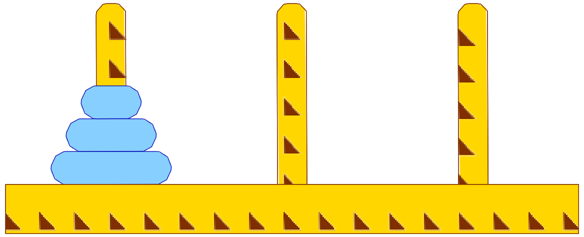


Les tours de Hanoï : algorithme

Tours de Hanoï
entrée : *jeu avec 1 pile de **n** disques (correctement ordonnés) et de 3 piliers A,B,C:* ***n*, *départ*, *transition*, *destination***
sortie : *jeu avec 1 pile de n disques (correctement ordonnés) sur le pilier de destination*
Si n > 0   **Tours de Hanoï** — entrée : ***n−1*, *départ*, *destination*, *transition***   ***Déplace 1 disque de départ à destination***   **Tours de Hanoï** — entrée : ***n−1*, *transition*, *départ*, *destination***

Démo avec 4 disques : http://upload.wikimedia.org/wikipedia/commons/6/60/Tower_of_Hanoi_4.gif

Les tours de Hanoï : exemple d'exécution



pilier: **A** **B** **C**

Hanoi (2, A,C,B)
- **A-> C**
- **A-> B**
- **C-> B**

Hanoi (3, A,B,C)

A -> C — **A -> C**

départ

transition

destination

Hanoi (2, B,A,C)
- **B-> A**
- **B-> C**
- **A-> C**

Sommes des n premiers entiers

Calculer la somme des *n* premiers entiers.

Si je peux le faire pour *n*, je peux le faire pour $n + 1$:

$$S(n+1) = (n+1) + S(n)$$

Mise en forme algorithmique récursive :

Si je veux résoudre le problème pour *n*,
Alors je cherche à l’exprimer à partir de la solution du problème plus simple pour n-1 :

$$S(n) = n + S(n-1)$$

Algorithme récursif

Le schéma général d'un algorithme récursif est le suivant :

monalgo_rec

entrée : *entrée du problème*
sortie : *solution du problème*

. . .

monalgo_rec

entrée : entrée du sous-problème
sortie : sol. du sous-problème

. . .

. . .

Exemple (incomplet) :

somme

entrée : n
sortie : $S(n)$

somme

entrée : $n - 1$
sortie : m

Sortir : $n + m$

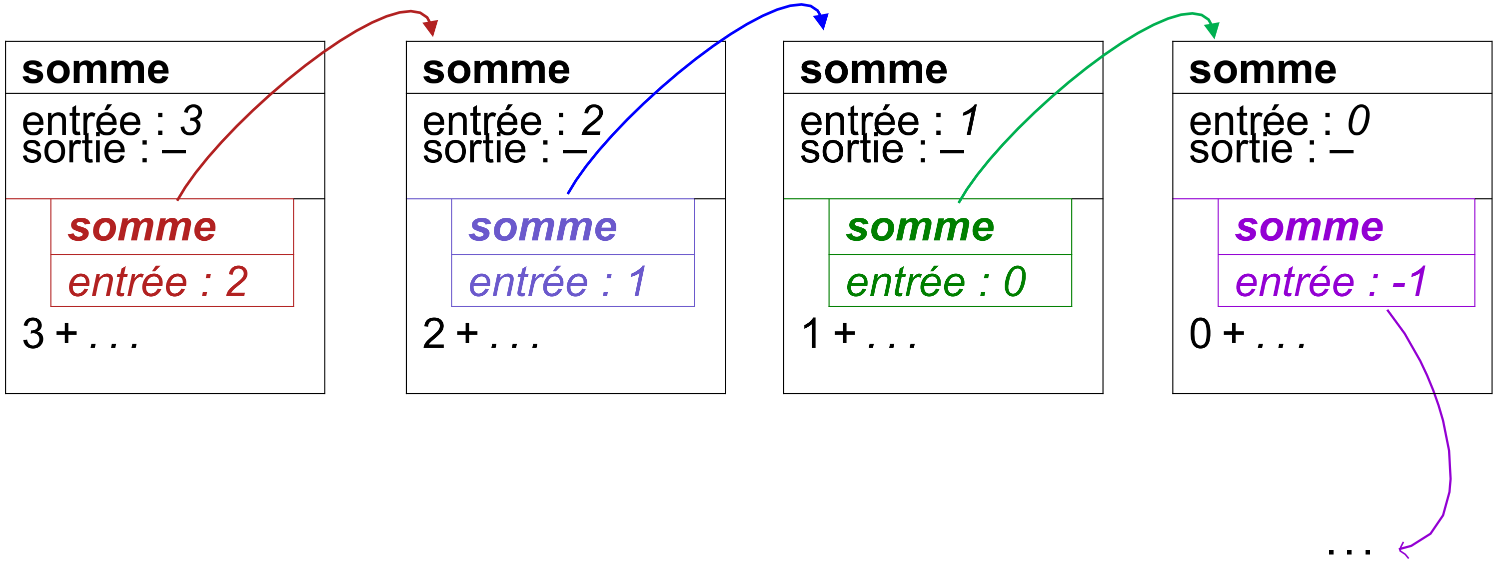
Condition de terminaison

Attention ! Pour que la résolution récursive soit correcte, il faut une

condition de terminaison

sinon, on risque une boucle infinie.

Exemple :



Algorithme récursif (correct)

Le schéma général correct d'un algorithme récursif est donc le suivant :

monalgo_rec

entrée : X
sortie : Y

si ***terminaison(X)*** ***alors*** ***Sortir*** *: . .*
sinon

. . .

> ***monalgo_rec***
>
> *entrée : entrée de l'instance réduite*
> *sortie : . . .*
>
> *. . .*

. . .

Somme récursive

Reprenons la somme des *n* premiers entiers positifs :

somme
entrée : n sortie : $S(n)$
si $n \leq 0$ Sortir: 0 ***sinon*** <table><tr><td>**somme**</td></tr><tr><td>*entrée :* $n - 1$ *sortie :* m</td></tr></table>   Sortir: $n + m$

Somme récursive: appels récursifs

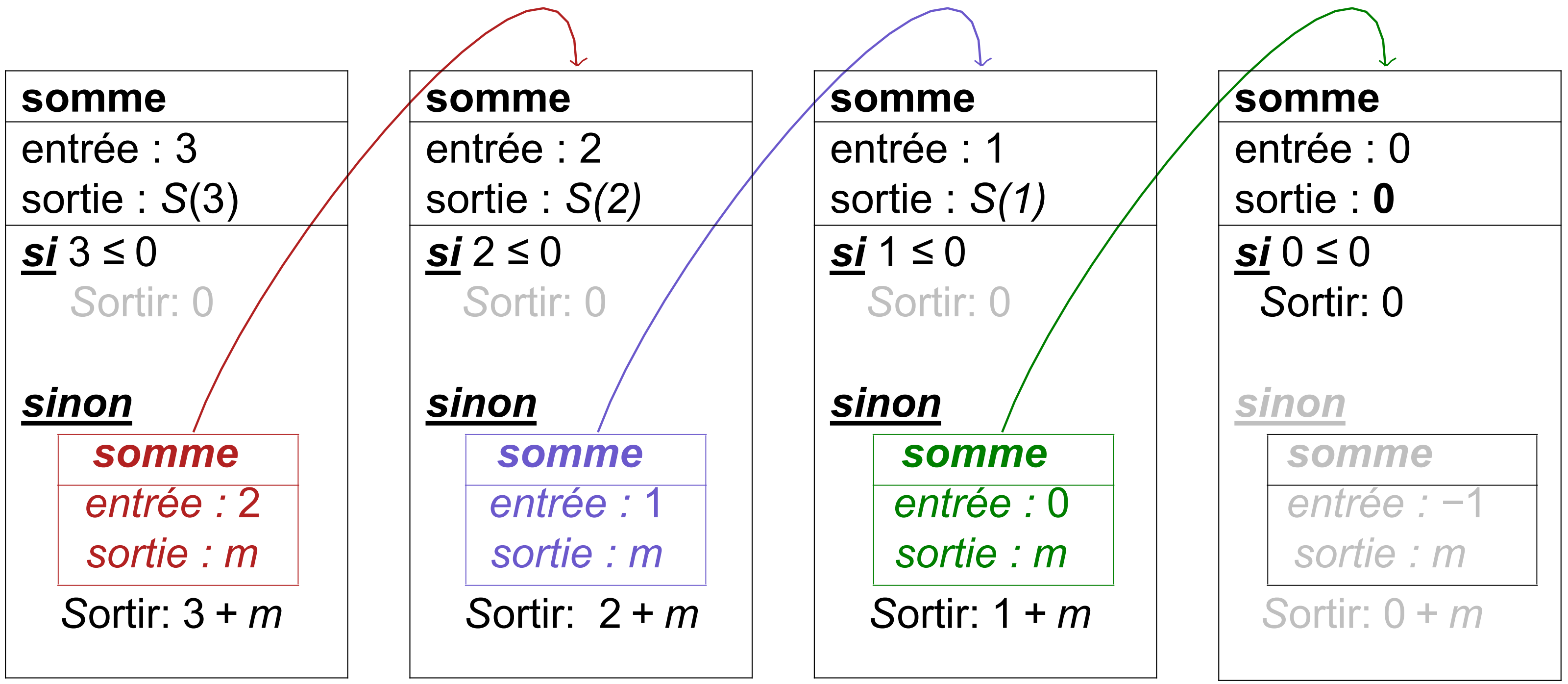
somme
entrée : 3
sortie : $S(3)$
si $3 \leq 0$
Sortir: 0
sinon
somme
entrée : 2
sortie : m
Sortir: $3 + m$

somme
entrée : 2
sortie : $S(2)$
si $2 \leq 0$
Sortir: 0
sinon
somme
entrée : 1
sortie : m
Sortir: $2 + m$

somme
entrée : 1
sortie : $S(1)$
si $1 \leq 0$
Sortir: 0
sinon
somme
entrée : 0
sortie : m
Sortir: $1 + m$

somme
entrée : 0
sortie : $S(0)$
si $0 \leq 0$
Sortir: 0
sinon
somme
entrée : −1
sortie : m
Sortir: $0 + m$

Somme récursive: retours(1)



Somme récursive: retours(2)

somme	**somme**	**somme**
entrée : 3 sortie : *S*(3)	entrée : 2 sortie : *S(2)*	entrée : 1 sortie : **1**
si 3 ≤ 0 Sortir: 0 ***sinon*** **somme** *entrée : 2* *sortie : m* Sortir: 3 + *m*	***si*** 2 ≤ 0 Sortir: 0 ***sinon*** **somme** *entrée : 1* *sortie : m* Sortir: 2 + *m*	***si*** 1 ≤ 0 Sortir: 0 ***sinon*** **somme** *entrée : 0* *sortie :* 0 Sortir: 1 + 0 = 1

Somme récursive: retours(3)

somme	**somme**
entrée : 3 sortie : $S(3)$	entrée : 2 sortie : ***3***
si $3 \leq 0$ Sortir: 0 ***sinon*** ***somme*** *entrée : 2* *sortie : m* Sortir: 3 + *m*	***si*** $2 \leq 0$ Sortir: 0 ***sinon*** ***somme*** *entrée : 1* *sortie : 1* Sortir: 2 + 1 = 3

Somme récursive: retours(3)

somme

entrée : 3
sortie : ***6***

si 3 ≤ 0
Sortir: 0

sinon

somme
entrée : 2
sortie : 3

Sortir: 3 + 3 = 6

S(3) renvoie 6 comme résultat

Somme récursive: remarques

Il est souvent plus efficace d'écrire la fonction sous une autre forme que la forme récursive.

Exemple de la somme des *n* premiers entiers :

$$S(n) = n + S(\ n\text{-}1\ )$$

mais on a aussi : $S(n) = \sum_{i=1}^{n} i$

s ← 0
Pour i de 1 à n
s ← s + i

Si elle existe, l'idéal est d'utiliser une expression analytique (pourquoi?):

$$S(n) = \frac{n\,(n+1)}{2}$$

Pour conclure temporairement sur la récursion

La solution récursive n'est pas toujours la seule solution et rarement la plus efficace...

...mais elle est parfois beaucoup **plus simple** et/ou **plus pratique** à mettre en œuvre !

Exemples : tris, traitement de structures de données récursives (e.g. arbres, graphes, ...), ...

SpeakUp M1.L4: what is printed by the recursive algorithm AlgoRec (n) for n = 13

AlgoRec
entrée : n, entier naturel sortie : aucune
Si n > 1 **AlgoRec(**n/2**)** //division entière Afficher (n mod 2)

A: 1010 B: 1011

C: 1001 D: 1101

Plan

- Stratégie de conception récursive
  - Les tours de Hanoï
  - Somme des n premiers entiers
- Stratégie de conception «Diviser pour règner»
  - Le tri fusion
- Stratégie de conception par programmation dynamique
  - Coefficient binomial
  - Le plus court chemin

Divide and Conquer

Pour un problème *P* portant sur un ensemble de **données** **d**,
le schéma général de l'approche « *diviser pour régner* » est le suivant :

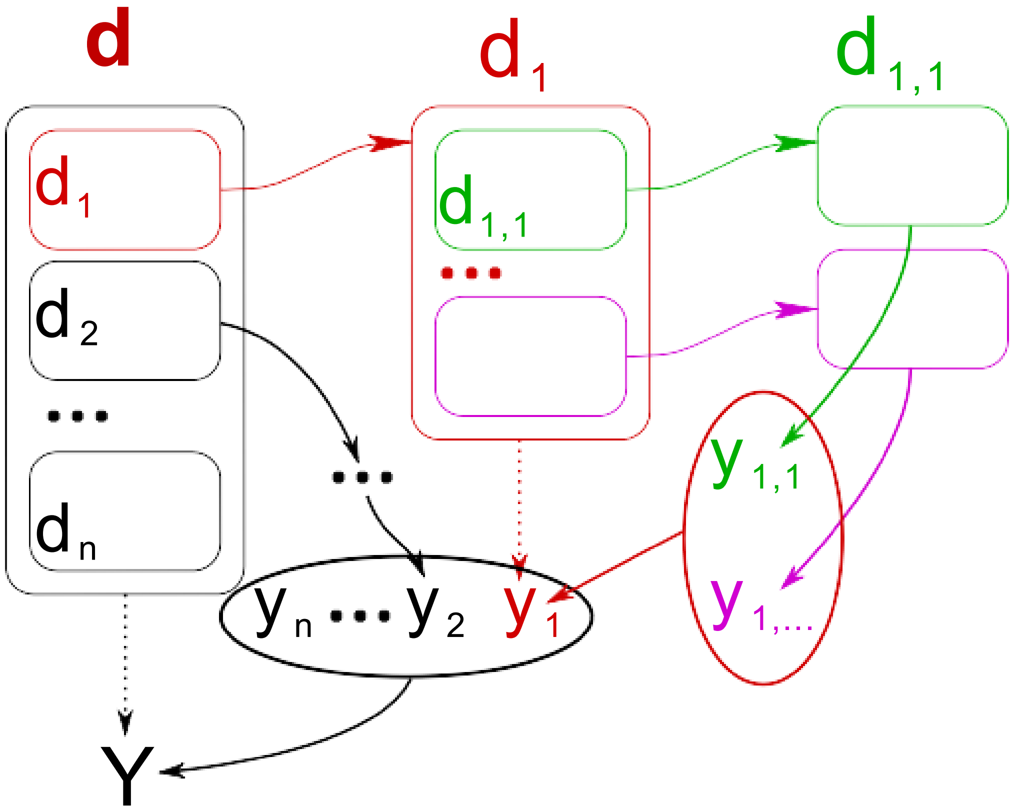
- si **d** est suffisamment simple, on peut résoudre facilement le problème (cas triviaux)

- Sinon,
  - décomposer **d** en instances plus petites $d_1, ..., d_n$
  - puis pour chacun des d_i : résoudre $P_i(d_i)$. On obtient alors une solution y_i
  - recombiner les y_i pour former la solution **Y** au problème de départ.

☞ conduit souvent à des algorithmes récursifs

Divide and Conquer

Pour un problème *P* portant sur des **données** *d*,
le schéma général de l'approche « *diviser pour régner* » est le suivant :



Tri Fusion

Donnée : soit une liste non vide **L** de taille n.
Produire une copie triée L' de L**.**

Principe (ébauche du pseudocode):

Si la taille de **L** est de 1 ou 2 éléments,
Le tri est effectué car trivial
(swap = échange à coût constant)
Sinon on calcule l'indice du milieu de **L**
l'algorithme s'appelle récursivement sur
les ***sous-listes*** gauche et droite.
*//chaque appel récursif renvoie une **copie** triée*
// de la sous-liste reçue en entrée

On utilise alors un algorithme **fusion_listes** qui prend en entrée les 2 sous-listes triées ***Left*** et ***Right*** et renvoie en sortie une liste triée de taille n.

tri_fusion

entrée : *Liste non vide **L**, taille **n** de la liste*
*Sortie : Liste **L'** triée de taille **n***

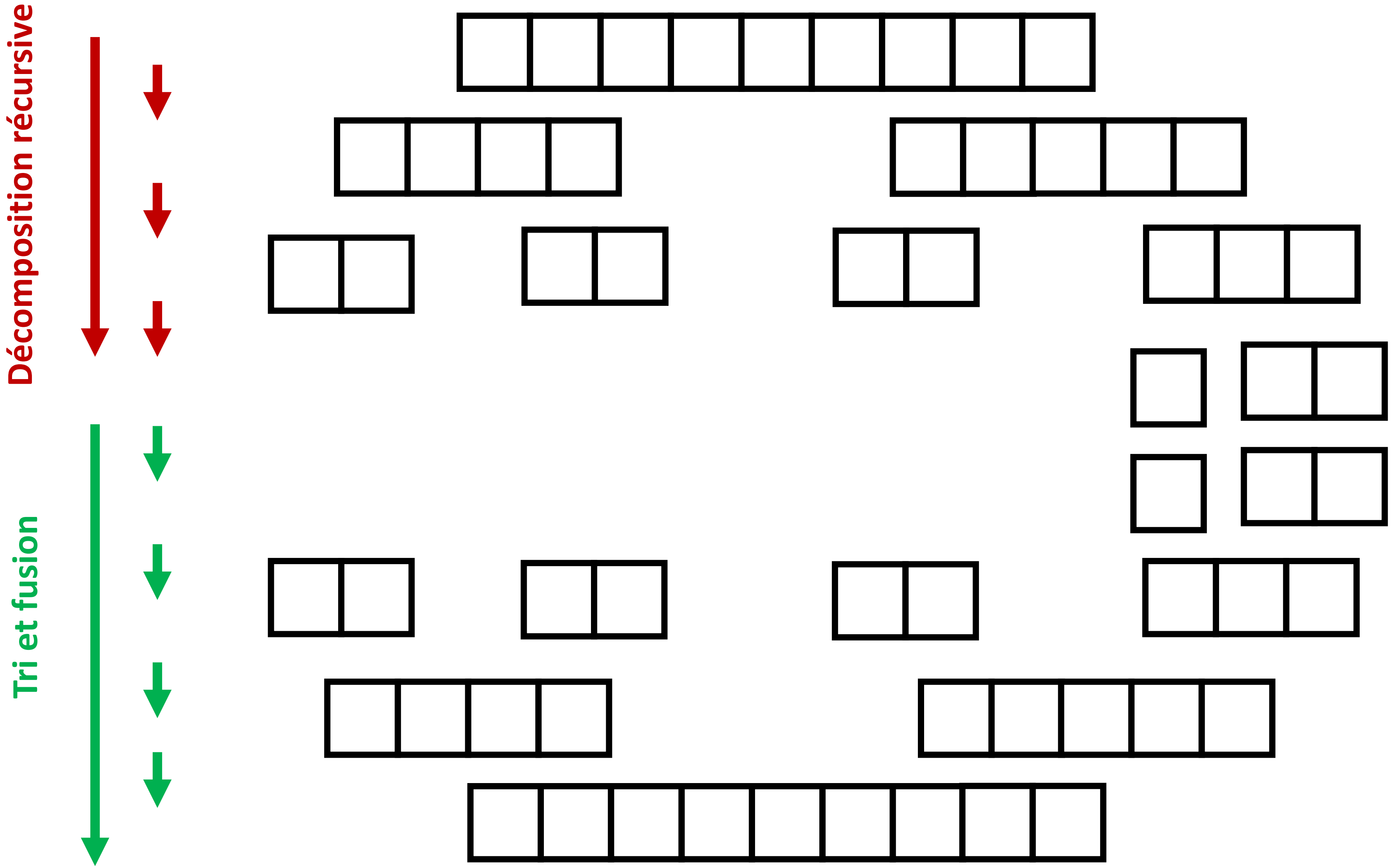
```
L' ← L
Si n = 1
    Sortir: L'

Si n = 2
    Si (L'(1) > L'(2))
        swap(L'(1), L'(2))
    Sortir: L'

mid ← n/2
Left ← tri_fusion(L(1 à mid), n/2)
Right ← tri_fusion(L(mid+1 à n), n/2 +n%2)

Sortir : fusion_listes (Left, n/2, Right, n/2 +n%2)
```

Exemple: le tri_fusion (merge sort)



Fusion de 2 listes triées dans une liste unique triée

Donnée : En entrée on a deux listes non vides Left et Right, respectivement de taille N1 et N2, qui sont déjà triées dans l'ordre croissant. On désire construire en sortie la liste F de taille n=N1+N2 qui fusionne les deux listes en respectant l'ordre croissant.

Principe (ébauche du pseudocode):
Remplir la liste fusionnée F, élément par élément, en choisissant le plus petit élément des deux listes fournies en entrées.

Cela est possible tant qu'il reste quelque chose à consommer dans les deux listes Left et Right.

Dès qu'une liste est vide, il suffit de copier le reste de l'autre liste.

Question: Quel est son ordre de complexité en fonction de n ?

Fusion_listes
entrées : *Liste non vide **Left**, taille N1* *Liste non vide **Right**, taille N2* sortie : *Liste **F** de taille n = N1+N2*

n ← N1 + N2
i ← 1, j ← 1, k ← 1
Tant que i ≤ N1 **et** j ≤ N2
    Si Left (i) < Right (j)
        F(k) ← Left (i)
        i ← i + 1
    Sinon
        F(k) ← Right (j)
        j ← j+ 1
    k ← k+ 1
 	
Pour pos de i à N1
    F(k) ← Left (pos)
    k ← k + 1
 	
Pour pos de j à N2
    F(k) ← Right (pos)
    k ← k + 1
Sortir: F

Exemple d’exécution de fusion_listes

Exemple: Left = { 4,6,9} et Right = {3,5,10}, n prend la valeur 6 qui sera la taille de F

Construction progressive de F avec l’algo fusion:

valeurs de:

i	
j	
k	
F(k)	
pos	

Left

4	6	9

i : 1 2 3

Right

3	5	10

j : 1 2 3

F

k : 1 2 3 4 5 6

EPFL

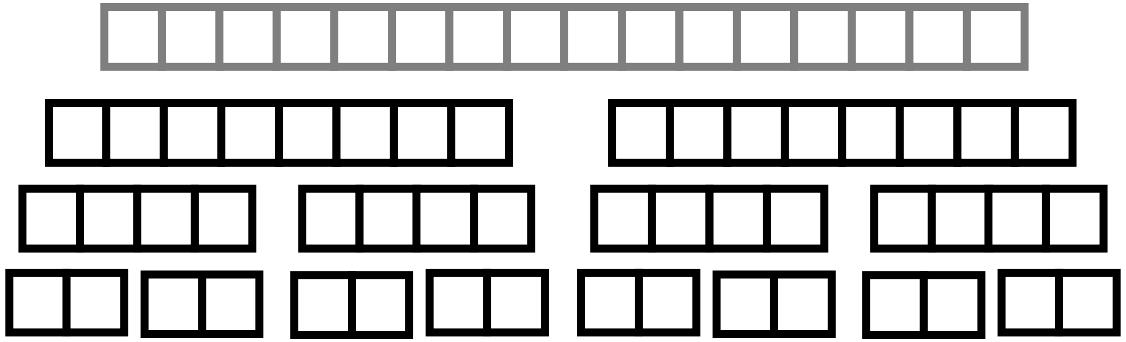
Tri Fusion : ordre de complexité

Éléments de l'algorithme à analyser pour une liste de taille **n** en entrée

// traitement des cas de terminaison	**Constant O(1)**
// divide mid ←n/2	**Constant O(1)**
// conquer Left ← **tri_fusion**(L(1 à mid), n/2) Right ← **tri_fusion**(L(mid+1 à n), n/2+n%2)	O(nlog(n))
// fusion des 2 sous-listes *Sortir :* **fusion_listes** (Left, n/2, Right, n/2+n%2)	**Linéaire O(n)**

Analyse de l'étape «conquer» :

Profondeur	Entrée: taille n
1	2^1 x n/ 2^1 → n
2	2^2 x n/ 2^2 → n
3	2^3 x n/ 2^3 → n



A chaque profondeur, le coût calcul est la somme des coût de fusion des sous-listes: leur nombre 2 fois plus grand est compensé par leur taille 2 fois plus petite. **Bilan: O(n) par niveau de profondeur.**

Combien de niveau de profondeurs faut-il compter ?

-> de l'ordre de $\log_2(n)$ pour atteindre le critère de terminaison. **Bilan: O(nlog$_2$(n))**

Plan

- Stratégie de conception récursive
  - Les tours de Hanoï
  - Somme des n premiers entiers
- Stratégie de conception «Diviser pour règner»
  - Le tri fusion
- Stratégie de conception par programmation dynamique
  - Coefficient binomial
  - Le plus court chemin

Programmation dynamique

La **programmation dynamique** est une méthode de résolution permettant de traiter des problèmes ayant une structure séquentielle répétitive.

« problèmes séquentiels » : pour lesquels on doit faire un ensemble de choix *successifs*/prendre des décisions *successives* pour arriver à une solution ; au fur et à mesure que de nouvelles options sont choisies, des sous-problèmes apparaissent (aspect « séquentiel »).

☞ La programmation dynamique s'applique lorsqu'un *même sous-problème* apparait dans *plusieurs* sous-solutions différentes.

Le principe est alors de stocker la solution à chaque sous-problème au cas où il réapparaitrait plus tard dans la résolution du problème global :

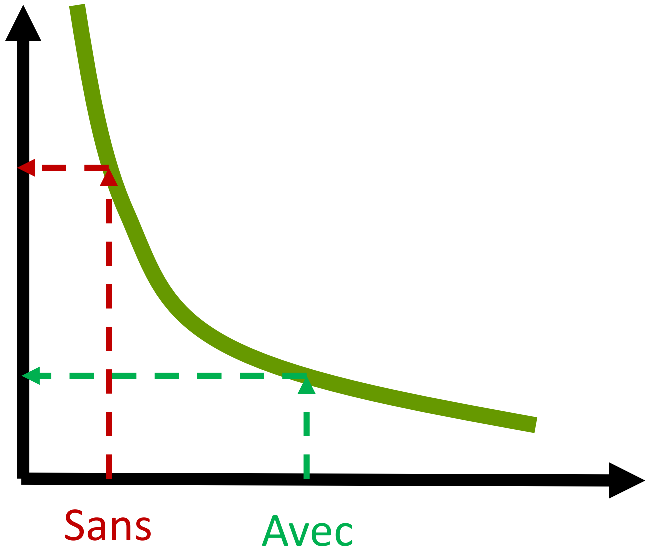
On évite de calculer plusieurs fois la même chose.

Programmation dynamique (2)

La programmation dynamique est souvent utilisée lorsque une solution récursive se révèle inefficace.

Elle permet souvent de changer un algorithme « naïf » coûteux en un algorithme, peut être plus complexe à concevoir, mais plus efficace.

Coût calcul de l'algorithme = nb d'opérations pour obtenir le résultat



Coût mémoire de la structure de donnée
= espace mémoire nécessaire
pour les données fournies en input,
le résultat *ET les calculs intermédiaires*

Exemple d'une solution récursive inefficace

Prenons l'exemple du calcul récursif des coefficients du binôme $\binom{n}{k}$

Problème C(n,k) :

Entrée : n, entier positif (ou nul) et k entier positif (ou nul), $k \leq n$.

Sortie : $\binom{n}{k}$

Rappel (formule de Pascal) :

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

Approche récursive :

- si $k = 0$ ou $k = n$,

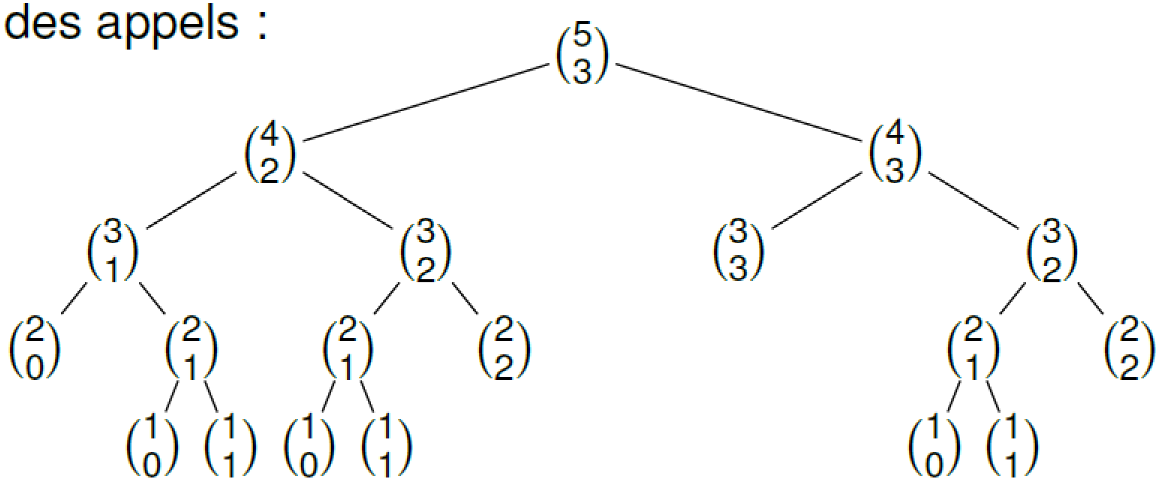
  Sortir: 1

- sinon

  Sortir: **C(n – 1, k – 1) + C(n – 1, k)**

Coefficients du binôme: approche récursive

Schéma des appels :



Quelle est la complexité de cette approche ?

Cas où l’arbre des appels récursifs est relativement «équilibré» : **k = (n/2) +1**

Ce scénario **multiplie par deux** le nombre d’appels résursifs jusqu’à une **profondeur de n/2**

☞ Le nombre d’appels récursifs est au minimum de $\mathbf{2^{(n/2)}}$
ce qui produit un ordre de complexité exponentiel en fonction de ***n***

Coefficients du binôme: approche récursive (2)

Y'a-t-il une meilleure solution ?

Idée : **ne pas recalculer plusieurs fois la même chose**

☞ stocker dans un **tableau** les valeurs déjà calculées et utiles pour la suite.

☞ Concrètement ici : le triangle de Pascal

Coefficients du binône par programmation dynamique

Tabuler les valeurs déjà calculées dans une table à deux indices, de taille (n+1)x(k+1) :

j

i	0	1	…	k
0	1			
1	1	1		
	1	2	1	
…	1	3	3	1
n	…			$\binom{n}{k}$

la table ne va pas au-delà de la colonne k+1

Calcul par programmation dynamique du coefficient $\binom{n}{k}$:

table(i,0) = table(i, i) = 1 pour i ≤ k

Pour les autres valeurs de **j < k** de la ligne d'indice **i**, il suffit de remplir la table en utilisant les valeurs mémorisées de la ligne d'indice **i-1 :**

table(i,j) = table(i-1, j-1) + table(i-1, j)

Quelle est la complexité de cet algorithme ? (~ nb d'éléments du triangle)

Coefficients du binône par programmation dynamique (2)

$$\begin{array}{cc|cccc} & & & & j & \\ & & 0 & 1 & \dots & k \\ \hline & 0 & 1 & & & \\ & 1 & 1 & 1 & & \\ i & & 1 & 2 & 1 & \\ & \dots & 1 & 3 & 3 & 1 \\ & n & \dots & & & \binom{n}{k} \end{array}$$

Quelle est la complexité de cet algorithme ?
nb d'éléments du triangle de Pascal:

= Somme des entiers de 1 à (n+1) :

1 + 2 + 3 + 4 + . . . + (n-1) + n + (n+1)

n + 2

1 + (n+1) = n+2

On peut construire (n+1)/2 paires dont la valeur est n+2

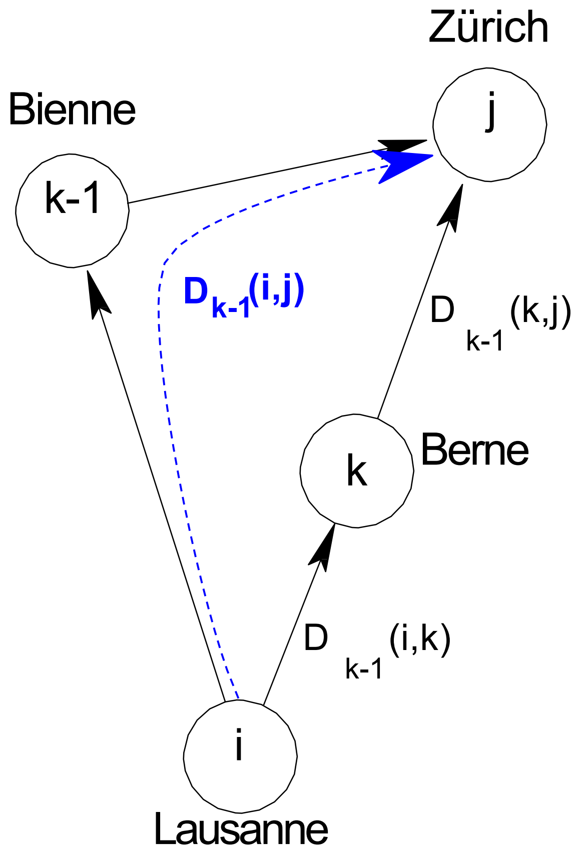
Somme des entiers de 1 à n+1 = (n+2)*(n+1)/2

Terme dominant en n^2 ; cette approche est en $O(n^2)$ au lieu d'un coût exponentiel

Programmation Dynamique – Autre exemple

Calcul du **plus court chemin**, par exemple entre **n** gares du réseau CFF

Voyons une solution par programmation dynamique :



Algorithme de Floyd

a) Initialiser la table **D** des **n x n** distances *connues* entre **n** gares

D(i, j) est la distance pour aller de la **gare i** à la **gare j**

b) **Pour** chaque gare **k**, de **1** à **n**
Mettre à jour chaque distance **D(i, j)** de la table en comparant:

1) la meilleure distance déjà présente dans la table **D(i, j)**

2) La distance obtenue pour aller de **i** à **j** en passant par **k**

D(i, k) + D(k, j)

$$D_k(i,j) = \min\{ D_{k-1}(i,j),\ D_{k-1}(i,k) + D_{k-1}(k,j) \}$$

Programmation Dynamique
Autre exemple (2)

L'algorithme est donc le suivant, **pour n gares** dans le réseau :

Initialisation d'une table des distances entre chaque paire de villes :

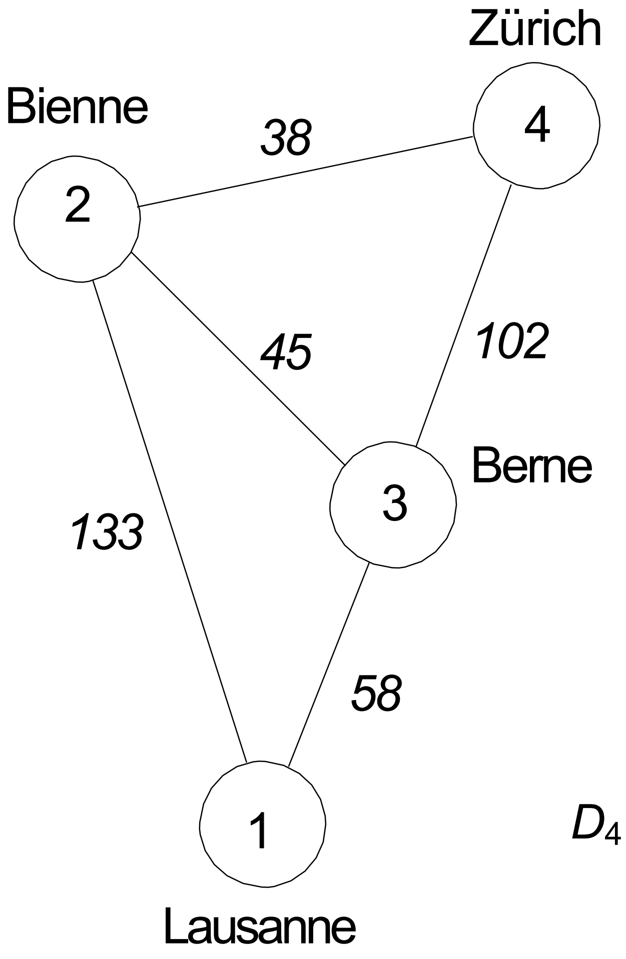
Pour *i* de 1 à *n*
 Pour *j* de 1 à *n*
  $D(i,j) \leftarrow$ distance *directe* de *i* à *j*, **+∞** si *i* et *j* ne sont pas directement connectés

Déroulement :

Pour ***k*** de 1 à *n*
 Pour *i* de 1 à *n*
  Pour *j* de 1 à *n*
   $D(i,j) \leftarrow \min\{\ D(i,j)\ ,\ D(i,k) + D(k,j)\ \}$

Combien d'exécution de cette instruction ?
☞ n^3

Algorithme de Floyd : exemple (valeurs km non réelles)



(données fictives)

$D_1 = D_0 =$

	Lausanne	Bienne	Berne	Zürich
Lausanne	0	133	58	∞
Bienne	133	0	45	38
Berne	58	45	0	102
Zurich	∞	38	102	0

$D_2 =$

	Lausanne	Bienne	Berne	Zürich
Lausanne	0	133	58	171
Bienne	133	0	45	38
Berne	58	45	0	83
Zurich	171	38	83	0

$D_4 = D_3 =$

	Lausanne	Bienne	Berne	Zürich
Lausanne	0	103	58	141
Bienne	103	0	45	38
Berne	58	45	0	83
Zurich	141	38	83	0

Note : fonctionne aussi pour des graphes asymétriques (graphes orientés)

Algorithmes de plus court chemin

L'algorithme de Floyd présenté ici résout en $O(n^3)$ étapes le problème du plus court chemin entre toutes les paires de gares

En appliquant le même genre d'idées (programmation dynamique) :

- l'algorithme de **Dijkstra** résout en $O(n^2)$ le problème du plus court chemin entre une gare donnée et toutes les autres

- l'algorithme **A*** (« *A star* ») est une généralisation de l'algorithme de Dijkstra qui est plus efficace si l'on possède un moyen d'estimer une borne inférieure de la distance <u>*restant à parcourir*</u> pour arriver au but

- ...et il existe plein d'autres algorithmes en fonctions des conditions spécifiques (graphe orienté/non orienté, coût positifs ou quelconques, graphe à cycles ou sans cycle)

Conclusion (1)

Formalisation des données : **structures de données abstraites**

Formalisation des traitements : **algorithmes**

☞ trouver des solutions correctes et distinguer formellement les solutions efficaces de celles inefficaces

Problèmes typiques : recherche, tris, plus « court » chemin.

La **conception** d’une méthode de résolution automatisée d’un problème consiste à choisir les *bons algorithmes* **et** les *bonnes structures de données*.

Conclusion (2)

La **conception** d'une méthode de résolution automatisée d'un problème consiste à choisir les *bons algorithmes* **et** les *bonnes structures de données*.

☞ Il n'y a pas de recette miracle pour cela, mais il existe des grandes familles de stratégies de résolution :

- **décomposer la structure du problème en sous-problèmes**: par une **analyse top-down** pour essayer de résoudre le problème en le décomposant en instances plus simples
- **décomposer les données en ensembles plus simples** pour lesquels la résolution du problème est triviale **« Divide and Conquer ».**
- Une mise en oeuvre récursive est souvent possible (mais pas obligatoire).
- **regrouper** (« programmation dynamique ») : mémoriser les calculs intermédiaires pour éviter de les effectuer plusieurs fois