

Information, Calcul et Communication

Module 1 : Calcul

Leçon 1.2 : Calcul et Algorithmes II

J.-C. Chappelier & J. Sam

Objectifs de la leçon

La leçon précédente a présenté ce qu'est un algorithme et par quels moyens l'exprimer.

Mais reste la principale question :

comment concevoir un algorithme

permettant de résoudre un problème donné ?

L'objectif de cette leçon est de vous présenter des *méthodes de résolution de problèmes* :

- ▶ « Diviser pour régner » (« *Divide and Conquer* »)
- ▶ Récursion
- ▶ Programmation dynamique

Conception d'algorithmes

Comment **concevoir** un algorithme permettant de résoudre un problème donné ?

Il n'y a malheureusement pas de méthode miracle ni de recette toute faite pour construire des solutions algorithmiques à un problème donné.

Une première démarche consiste à rechercher une ressemblance avec des problèmes déjà connus :

recherche

tri

plus court chemin

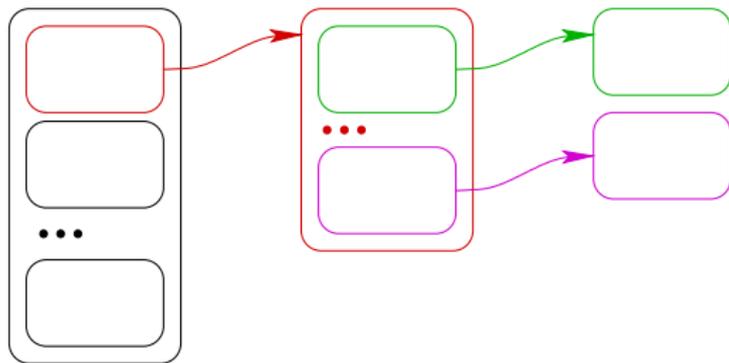
Sinon, il existe plusieurs **méthodes de résolution**, c'est-à-dire des *schémas d'élaboration de solutions*.

Plusieurs de ces méthodes suivent ce que l'on appelle **une approche descendante** (« *top-down* », procède par *analyse*), par opposition à ascendante (« *bottom-up* », procède par *synthèse*).

Approche descendante

Résoudre un problème par une **approche descendante** consiste à **décomposer** le problème général en **sous-problèmes** plus spécifiques, lesquels seront chacun décomposés en problèmes encore plus spécifiques, etc. (raffinements successifs)

Une telle analyse du problème se fait à l'aide de **blocs imbriqués** correspondant chacun à des résolutions de plus en plus **spécifiques**, décrites par des algorithmes de plus en plus spécialisés.



Exemple

Par exemple avec l'algorithme de tri par insertion vu à la leçon précédente :

On découpe le problème en sous-problèmes :

tri insertion
entrée : <i>un tableau (d'objets que l'on peut comparer)</i> sortie : <i>le tableau trié</i>
Tant que il y a un élément mal placé on cherche sa bonne place on déplace l'élément à sa bonne place

Chaque **sous-problème** étant ensuite spécifié plus clairement puis résolu.

Tri par insertion : résolution détaillée

Le sous-problème *rechercher un élément mal placé*

entrée : un tableau `tab`

sortie : position du 1^{er} élément strictement plus petit que son prédécesseur

La solution est ici assez simple :

On effectue une **itération** sur les éléments de `tab` en s'arrêtant au premier élément strictement plus petit que son prédécesseur.

Comme le 1^{er} élément de `tab` ne peut être mal placé (car sans prédécesseur), l'itération de recherche d'un élément mal placé commencera à partir du 2^e élément

De même, s'il n'y a pas d'élément mal placé on retournera, par convention, la position 1.

Tri par insertion : résolution détaillée (2)

Le sous-problème *trouver la bonne place*

entrée : un tableau `tab` et l'entier `pos`, position d'un élément mal placé

sortie : la bonne position `pos_ok` de l'élément mal placé.

La « bonne position » correspond à la plus grande position `pos_ok` ($< pos$) dans le tableau `tab` telle que le $(pos_ok - 1)$ -ième élément de `tab` soit inférieur ou égal au `pos`-ième.

L'algorithme pour *trouver la bonne place* doit donc parcourir les éléments de `tab`, un à un, entre le premier et celui à la position `pos`, à la recherche de la bonne position.

Cet algorithme effectue donc aussi une **itération** sur les éléments du tableau, du premier élément à celui de position `pos`.

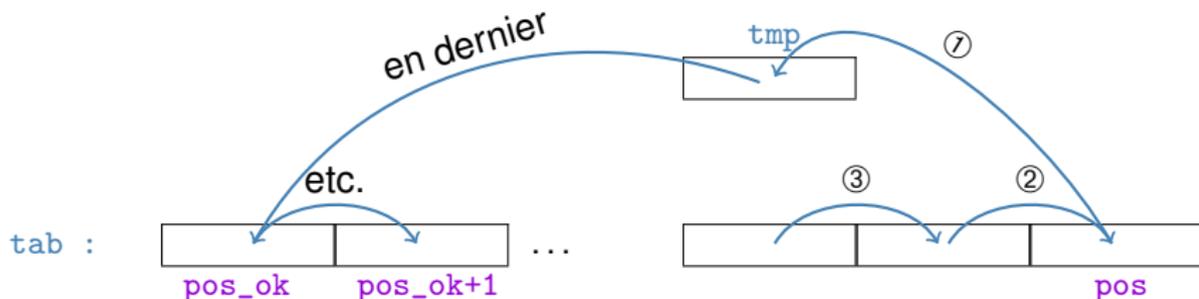
Tri par insertion : résolution détaillée (3)

Le sous-problème *déplacer un élément*

entrée : un tableau `tab`, une position de départ `pos` et une position finale `pos_ok`

On doit déplacer l'élément de la position `pos` dans `tab` à la position `pos_ok`.

On peut effectuer cette opération par **décalages successifs** (en utilisant un stockage temporaire `tmp`).



Synthèse

tri insertion

entrée : un tableau T

sortie : le tableau trié

$pos \leftarrow \mathbf{mal_placé}(T)$

Tant que $pos > 1$

$pos_ok \leftarrow \mathbf{bonne_place}(T, pos)$

$\mathbf{déplace}(T, pos, pos_ok)$

$pos \leftarrow \mathbf{mal_placé}(T)$

avec :

mal_placé

entrée : un tableau T

sortie : position du premier élément mal placé

...à vous de l'écrire...

etc.

Améliorations

1. Pour **rechercher le prochain élément mal placé**, ce n'est pas la peine de recommencer du début (position 2) à chaque fois. On peut partir de *la dernière position mal placée*.
 2. On pourrait **trouver la bonne place** et **déplacer l'élément** à cette place *en même temps* (i.e. en *une seule* itération)
- Si l'on regroupe tout ceci, on arrive à l'algorithme suivant :

```
Pour i de 2 à N (= taille du tableau)
| tmp ← tableau[i]
| j ← i
| Tant que j ≥ 2 et tableau[j-1] > tmp
| | tableau[j] ← tableau[j-1]
| | j ← j-1
| tableau[j] ← tmp
```

Divide and Conquer

Parmi les méthodes descendantes, une qui est souvent mise en œuvre s'appelle « **diviser pour régner** » (divide and conquer).

Elle consiste à **diviser/regrouper les données** pour résoudre des (sous-)problèmes plus simples.

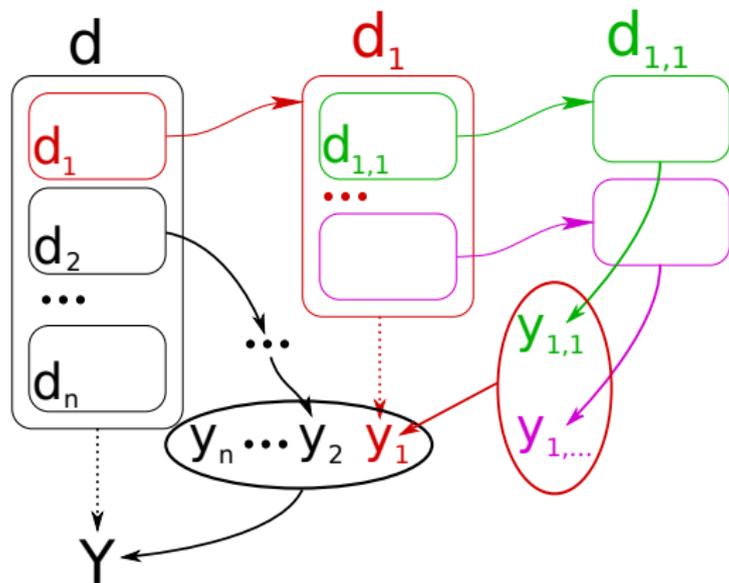
Cette idée n'est pas nouvelle :

« *Diviser chacune des difficultés que j'examinerois, en autant de parcelles qu'il se pourroit, et qu'il soit requis pour les mieux résoudre* »

(Descartes, *Discours de la méthode*, 17^e siècle)

Divide and Conquer

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



Divide and Conquer

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

- ▶ si d est « assez simple », appliquer un algorithme « *ad hoc* » permettant de résoudre le problème (traitement des cas triviaux)
 - ▶ sinon,
 - ▶ décomposer d en instances plus petites d_1, \dots, d_n
 - ▶ puis pour chacun des d_i : résoudre $P_i(d_i)$.
On obtient alors une solution y_i
 - ▶ recombinaison des y_i pour former la solution Y au problème de départ.
- ☞ conduit souvent à des algorithmes récursifs

Récursion

Une catégorie particulière de méthodes de résolution de problèmes sont les solutions **récur­sives**.

Le principe de l'approche récur­sive est de

ramener le problème à résoudre à un sous-problème, version simplifiée du problème d'origine.

Exemples :

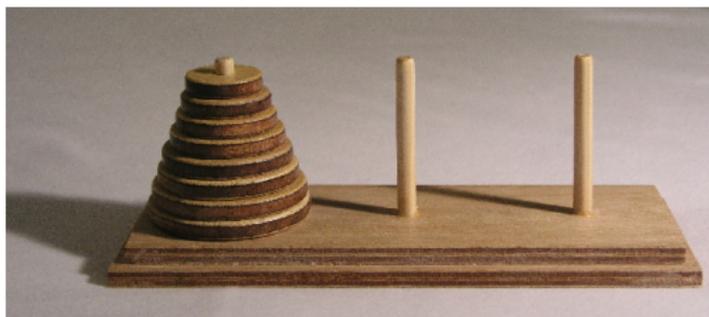
- ▶ recherche par dichotomie (cf leçon précédente)
- ▶ exemple en mathématiques : le raisonnement par récurrence
- ▶ les algorithmes dits récur­sifs (à suivre)

Exemple : Les tours de Hanoï

Jeu des tours de Hanoï :

déplacer d'un pilier à un autre une colonne de disques de taille croissante

- ▶ en utilisant un seul pilier de transition (c'est-à-dire 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 :

- ▶ 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 pile de n disques (correctement ordonnés) sur le pilier numéro i , $i, j (\neq i)$, nombre n de disques à déplacer

sortie : jeu avec pile de n disques (correctement ordonnés) sur le pilier numéro j

Si $n > 0$

Choisir k différent de i et j (par exemple $k = 6 - i - j$)

Tours de Hanoï

entrée : jeu, $i, k, n-1$

Déplace disque du pilier i au pilier j

Tours de Hanoï

entrée : jeu, $k, j, n-1$

démo : http://upload.wikimedia.org/wikipedia/commons/6/60/Tower_of_Hanoi_4.gif

Autre(s) exemple(s)

Calculer la somme des n premiers entiers.

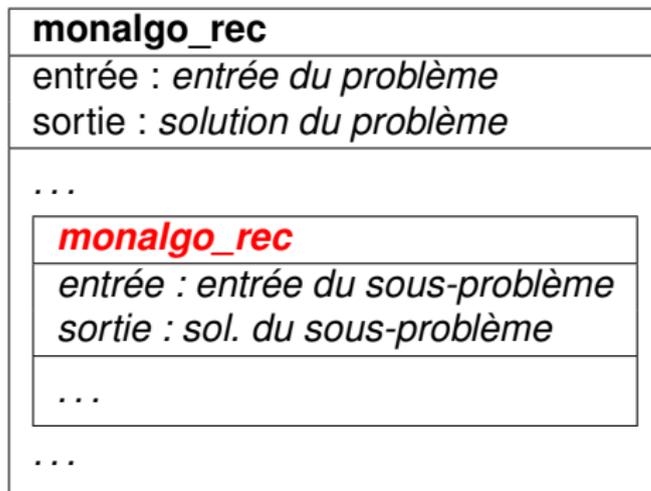
Si je peux le faire pour n , je peux le faire pour $n+1$:

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

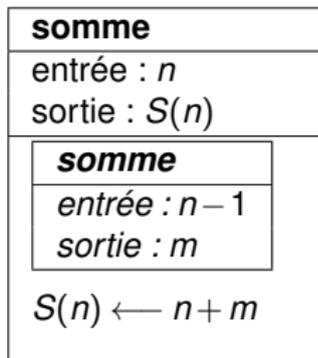
Note : se généralise trivialement au calcul de toute grandeur définie par une équation de récurrence.

Algorithme récursif

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



Exemple (incomplet) :



Condition de terminaison



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

condition de terminaison

sinon, on risque une boucle infinie.

Exemple :

somme
entrée : 3 sortie : -
somme entrée : 2
3 + ...

somme
entrée : 2 sortie : -
somme entrée : 1
2 + ...

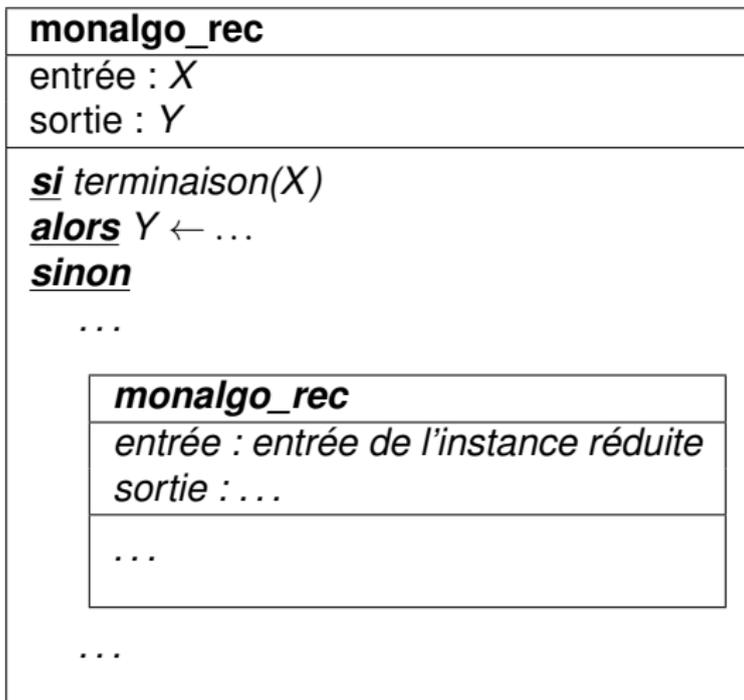
somme
entrée : 1 sortie : -
somme entrée : 0
1 + ...

somme
entrée : 0 sortie : -
somme entrée : -1
0 + ...

...

Algorithme récursif (correct)

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

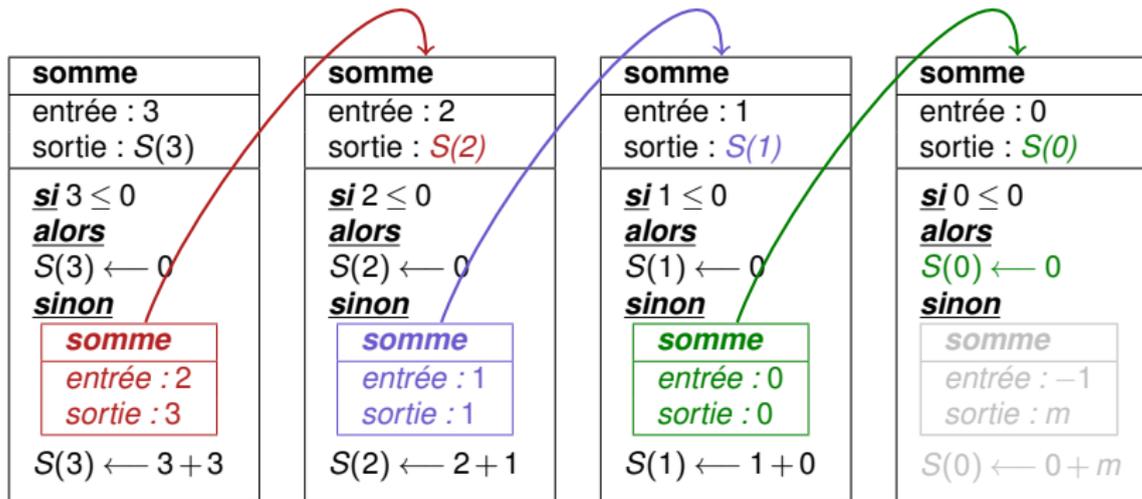


1^{er} exemple

Reprenons la somme des n premiers entiers positifs :

somme		
entrée : n sortie : $S(n)$		
<u>si</u> $n \leq 0$ <u>alors</u> $S(n) \leftarrow 0$ <u>sinon</u>		
<table border="1"><tr><td>somme</td></tr><tr><td>entrée : $n - 1$ sortie : m</td></tr></table>	somme	entrée : $n - 1$ sortie : m
somme		
entrée : $n - 1$ sortie : m		
$S(n) \leftarrow n + m$		

1^{er} exemple : déroulement



$$S(3) = 6$$

1^{er} exemple : remarques

Notez qu'il est parfois préférable d'écrire la fonction sous une autre forme que la forme récursive.

Si l'on reprend l'exemple de la somme des n premiers entiers :

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

mais on a aussi (!) :

$$S(n) = \sum_{i=1}^n i$$

(c'est-à-dire une itération) qui est plus direct que la forme récursive.

On peut parfois même utiliser une expression analytique (lorsqu'on en a une !); par exemple :

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

1^{er} exemple : complexités ?

- ▶ version récursive ?

somme		
entrée : n sortie : $S(n)$		
si $n \leq 0$ alors $S(n) \leftarrow 0$ sinon		
<table border="1"><tr><td>somme</td></tr><tr><td>entrée : $n - 1$ sortie : m</td></tr></table>	somme	entrée : $n - 1$ sortie : m
somme		
entrée : $n - 1$ sortie : m		
$S(n) \leftarrow n + m$		

- ▶ version itérative ($S(n) = \sum_{i=1}^n i$) ?

Exemple 2 : version récursive du tri par insertion

On peut aussi concevoir le tri par insertion de façon récursive :

tri
entrée : <i>tableau de n éléments</i> sortie : <i>tableau trié</i>
<i>condition arrêt : moins de 2 éléments</i>
<i>tri (instance réduite du problème)</i>
entrée : <i>tableau de $n - 1$ éléments</i> sortie : <i>tableau trié</i>
...
<i>insertion</i> du $n^{\text{ème}}$ élément dans le tableau trié de $n - 1$ éléments

Tri récursif : exemple

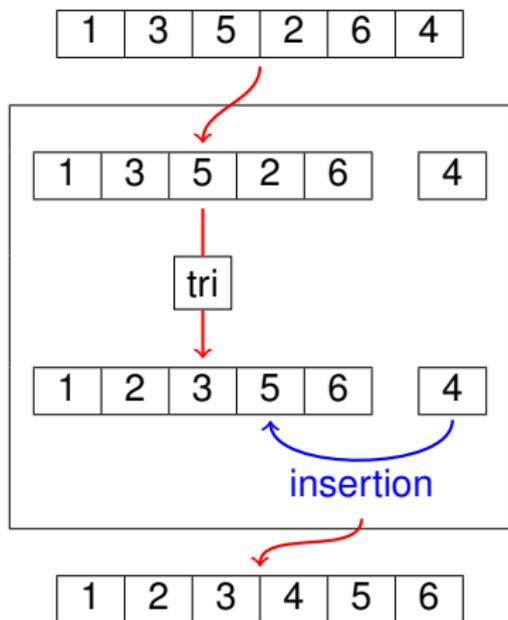
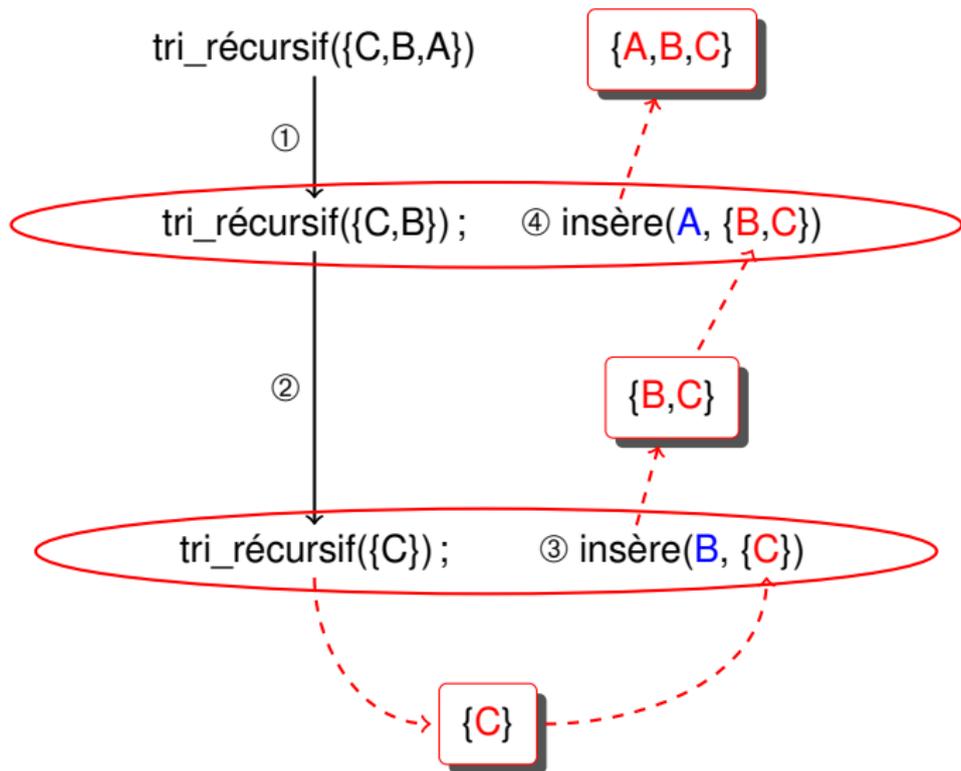


Schéma des appels récursifs (exemple)



Pour conclure 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, ...), ...

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* apparaît (avec les mêmes données) dans *plusieurs* sous-solutions différentes.

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

On évite de calculer plusieurs fois la même chose.

Note : cette idée (programmation dynamique) peut s'appliquer aussi bien à des approches descendantes qu'ascendantes.

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.

Exemple

Prenons l'exemple du calcul des coefficients du binôme $\binom{n}{k}$ (noté aussi C_n^k)

Problème $C(n, k)$:

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

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

Approche récursive :

- ▶ si $k = 0$ ou $k = n$, renvoyer 1
- ▶ sinon retourner $C(n-1, k-1) + C(n-1, k)$

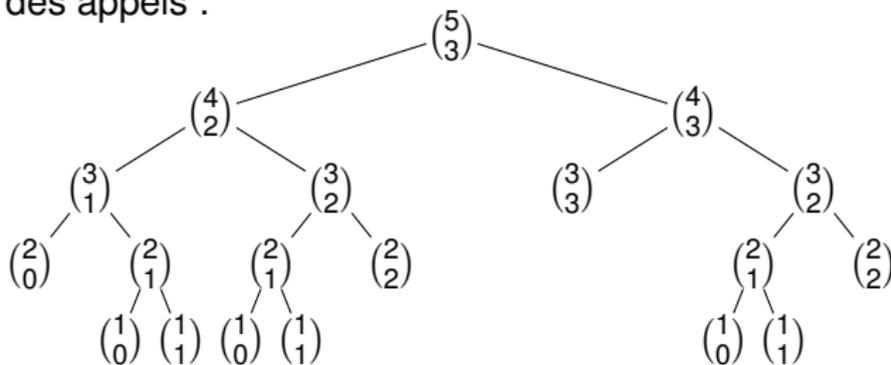
Rappel :

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

Coefficients du binôme

approche récursive

Schéma des appels :



Quelle est la complexité $T(k, n)$ de cette approche ?

Du fait de la récursion, on a :

(Supposons que les comparaisons et les additions soient des instructions élémentaires)

$$T(k, n) = T(k-1, n-1) + T(k, n-1) + 1$$

et d'autre part $T(0, 0) = 1$ et $T(n, n) = 1$

Coefficients du binôme approche récursive (2)

d'où :

$$T(k, n) = 2 \binom{n}{k} - 1$$

☞ temps **exponentiel** en fonction de n

Y'a-t-il une meilleure solution ?

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

(Regardez par exemple combien de fois nous avons calculé $\binom{1}{1}$!)

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

(on parle de *mémoïsation*/memoization)

Coefficients du binôme par programmation dynamique

« tabuler les valeurs déjà calculées »

☞ Concrètement ici : le triangle de Pascal :

The diagram shows Pascal's triangle with a vertical red arrow on the left labeled i pointing downwards to the value n , and a horizontal blue arrow at the top labeled j pointing to the right towards the value k . The triangle consists of the following rows of numbers:

	1			
	1	1		
	1	2	1	
	1	3	3	1
	1	4	6	4
n	1	5	10	10

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

- ▶ On remplit le début (k éléments) de chaque ligne du triangle de Pascal, une après l'autre, de haut en bas.
- ▶ On arrête à la ligne n .

Quelle est la complexité de cet algorithme ?

Programmation Dynamique – Autre exemple

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

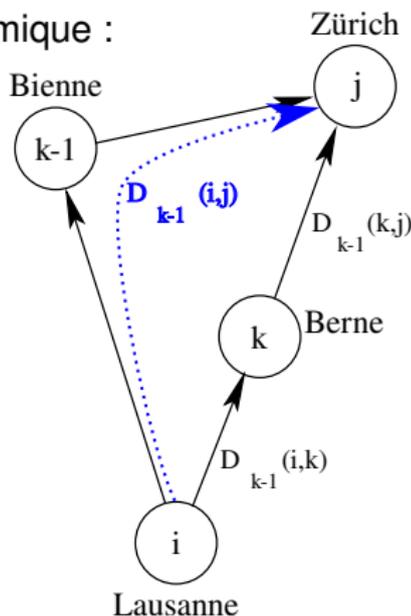
Voyons une solution par programmation dynamique :
l'Algorithme de Floyd

illustration de l'idée de base :

le plus court chemin pour aller de Lausanne à Zürich est le minimum entre :

1. le plus court chemin connu pour aller de Lausanne à Zürich,
2. le chemin allant de Lausanne à Zürich en passant par une ville intermédiaire non encore considérée.

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



Programmation Dynamique

Autre exemple (2)

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

Initialisation :

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

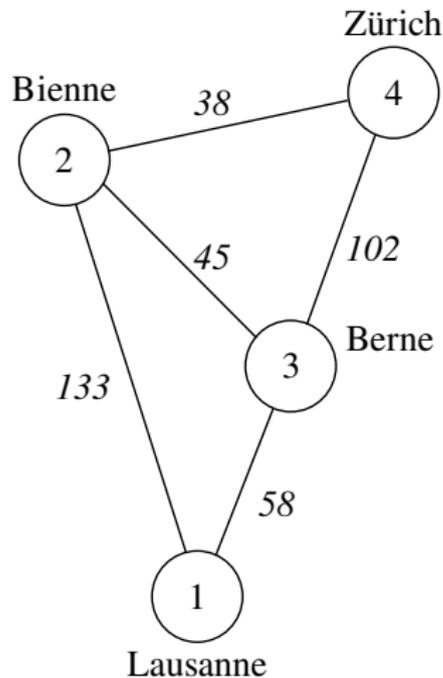
Pour j de 1 à n

$D(i,j) \leftarrow \min \{D(i,j), D(i,k) + D(k,j)\}$

Combien de boucles ?

↳ $\mathcal{O}(n^3)$

Algorithme de Floyd : exemple



$$D_1 = D_0 =$$

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

$$D_2 =$$

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

$$D_4 = D_3 =$$

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

(données fictives)

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

Algorithmes de plus court chemin

L'algorithme de Floyd présenté ici résout en $\mathcal{O}(n^3)$ 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 $\mathcal{O}(n^2)$ 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 restant à parcourir pour arriver au but (on appelle cela une « heuristique admissible » ; Dijkstra est un A* avec l'heuristique nulle)
- ▶ l'algorithme de Viterbi résout en $\mathcal{O}(n)$ le problème du plus court chemin entre deux gares données (sans cycle : DAG)
- ▶ ...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** (« Divide and Conquer ») : essayer de résoudre le problème en le *décomposant en instances plus simples*

Les algorithmes *récurifs* sont des illustrations de cette stratégie.

▶ **regrouper** (« programmation dynamique ») : *mémoriser les calculs intermédiaires* pour *éviter de les effectuer plusieurs fois*

La suite

- ▶ La prochaine leçon :
Qu'est-ce qui est calculable et ne l'est pas ?
- ▶ Puis :
Comment représenter l'information (les données sur lesquelles calculer) ?