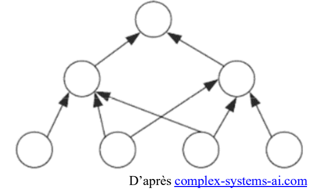
TNSI | Algorithmique — Programmation dynamique

Cours – Programmation dynamique

Objectifs :

- ➩ Découvrir une nouvelle stratégie algorithmique
- ➩ Comprendre l'intérêt de la programmation dynamique
- ➩ Voir quelques exemples d'application


D'après complex-systems-ai.com

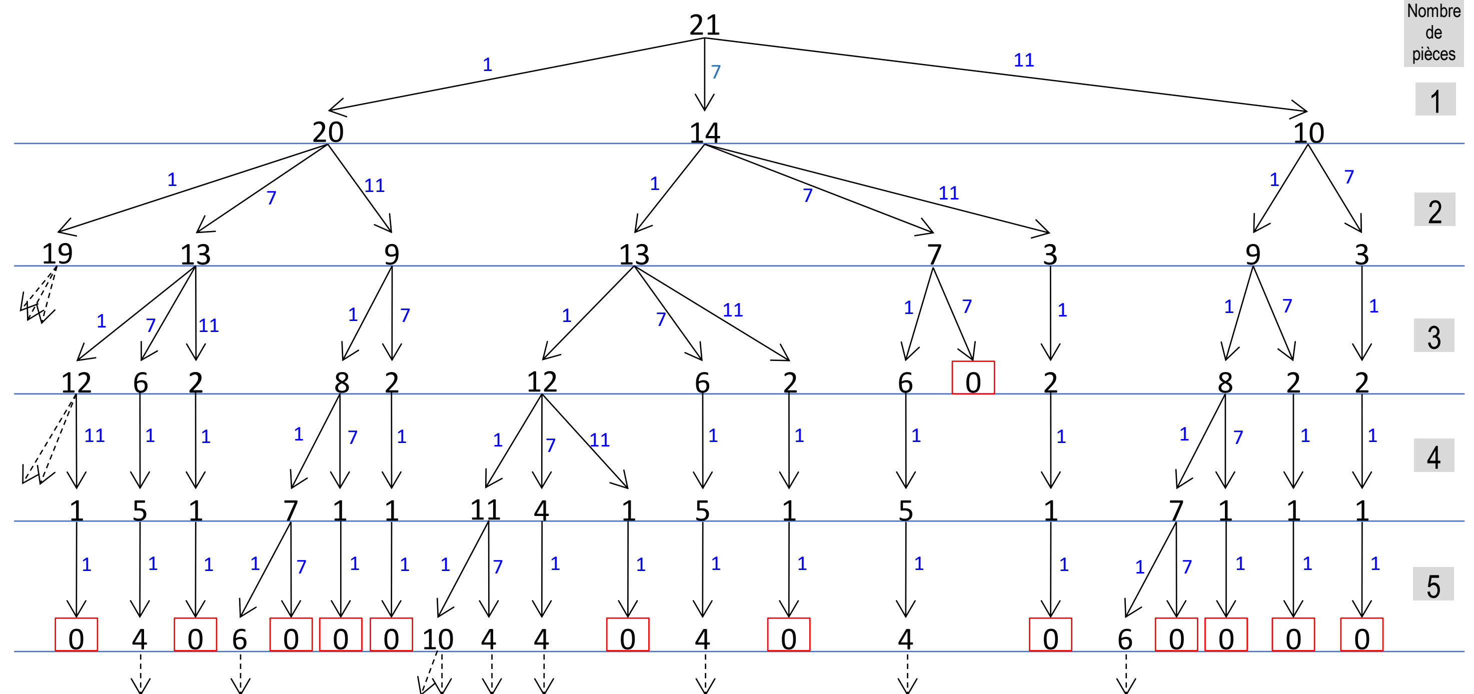
I – Le retour du rendu de monnaie

Nous avons déjà vu en première l'algorithme du rendu de monnaie :
Comment rendre la monnaie sur une somme donnée avec les pièces à disposition et en utilisant le moins de pièces possibles.
Nous avions utilisé un algorithme glouton pour trouver rapidement une solution au problème. Le souci c'est que cet algorithme ne trouve pas toujours la solution optimale.
Par exemple si on dispose de pièces de valeurs {1, 7, 11} et qu'on veut rendre la monnaie sur 21, l'algorithme glouton va nous proposer {11, 7, 1, 1, 1} soit 5 pièces alors qu'on pourrait faire {7, 7, 7} soit seulement 3 pièces.
Nous allons donc mettre au point un algorithme permettant de trouver à coup sûr la solution optimale.

Pour simplifier l'étude, on n'utilisera que des valeurs entières (on évite ainsi les soucis avec les égalités sur les flottants) et la valeur 1 sera toujours présente dans les pièces disponibles (il y a ainsi toujours une solution possible).

1) Approche top-down

L'idée est de partir de la somme à rendre et de réduire celle-ci en essayant les différentes pièces à disposition jusqu'à arriver à zéro.
A chaque nouvelle pièce, on va retenir la solution la meilleure (celle qui utilise le moins de pièces).



On part donc de la somme à rendre puis on essaye successivement et récursivement chaque pièce du stock (ici on essaye les valeurs de pièce 11, 7 puis 1). Quand on a essayé toutes les pièces possibles pour une étape, on ajoute à la solution la pièce qui donne le moins de pièces dans le rendu jusqu'ici.

Licence Creative Commons BY-SA

Le cheminement sur l'exemple précédent en examinant les pièces dans l'ordre {11, 7, 1} serait :

Nombre de pièces 1	Nombre de pièces 2	Nombre de pièces 3	Nombre de pièces 4	Nombre de pièces 5	Nombre de pièces 6	Meilleure solution
On essaye avec 11	Il reste 10 On essaye avec 7	Il reste 3 On essaye avec 1	Il reste 2 On essaye avec 1	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
	On essaye avec 1	Il reste 9 On essaye avec 7	Il reste 2 On essaye avec 1	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
		On essaye avec 1	Il reste 8 On essaye avec 7	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
			On essaye avec 1	Il reste 7 On essaye avec 7	Il reste 0 → Fini	6 pièces
				On essaye avec 1	Il reste 6 On essaye avec 1…	12 pièces
On essaye avec 7	Il reste 14 On essaye avec 7	Il reste 7 On essaye avec 7	Il reste 0 → Fini			3 pièces
		On essaye avec 1	Il reste 6 On essaye avec 1	Il reste 5 On essaye avec 1	Il reste 4 On essaye avec 1…	10 pièces
	On essaye avec 1	Il reste 13 On essaye avec 11	Il reste 2 On essaye avec 1	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
		On essaye avec 7	Il reste 6 On essaye avec 1	Il reste 5 On essaye avec 1	Il reste 4 On essaye avec 1…	10 pièces
		On essaye avec 1	Il reste 12 On essaye avec 11	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
			On essaye avec 7	Il reste 5 On essaye avec 1	Il reste 4 On essaye avec 1…	10 pièces
			On essaye avec 1	Il reste 11 On essaye avec 11	Il reste 0 → Fini	6 pièces
				On essaye avec 7	Il reste 4 On essaye avec 1…	10 pièces
				On essaye avec 1	Il reste 10 On essaye avec 7…	10 pièces
					On essaye avec 1…	10 pièces
						⋮
On essaye avec 1	Il reste 20 On essaye avec 11	Il reste 9 On essaye avec 7	Il reste 2 On essaye avec 1	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
		On essaye avec 1	Il reste 8 On essaye avec 7	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
			On essaye avec 1	Il reste 7 On essaye avec 7	Il reste 0 → Fini	6 pièces
				On essaye avec 1	Il reste 6 On essaye avec 1…	12 pièces
	On essaye avec 7	Il reste 13 On essaye avec 11	Il reste 2 On essaye avec 1	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
		On essaye avec 7	Il reste 6 On essaye avec 1	Il reste 5 On essaye avec 1	Il reste 4 On essaye avec 1…	10 pièces
		On essaye avec 1	Il reste 12 On essaye avec 11	Il reste 1 On essaye avec 1	Il reste 0 → Fini	6 pièces
			⋮			
	⋮					

<u>Application n°1 :</u>
Ecrire une fonction `rendu_monnaie_naif(somme, pieces)` qui prend en argument un entier `somme` correspondant à la somme à rendre et un tableau `pieces` contenant la liste des pièces disponibles (dans un ordre quelconque). Cette fonction reprendra l'algorithme récursif décrit précédemment.
Tester la fonction sur des exemples simples, par exemple `rendu_monnaie_naif(21, [11,7,1])` doit renvoyer `[7,7,7]`. `rendu_monnaie_naif(20, [11,7,1])` doit renvoyer `[11,7,1,1]`.

<u>Aide</u> : Si vous avez du mal à écrire l'algorithme, vous pouvez vous aider du pseudo-code qui suit.

Licence Creative Commons BY-SA

Algorithme 1 :

```
FONCTION rendu_monnaie_naif(somme_a_rendre, pieces)
    SI somme_a_rendre = 0 ALORS RENVOYER tableau vide
    plus_petit_nombre_de_pieces ← somme_a_rendre
    meilleure_solution ← tableau vide
    POUR CHAQUE piece dans pieces FAIRE
        SI somme_a_rendre ≥ piece ALORS
            solution ← rendu_monnaie_naif(somme_a_rendre – piece, pieces)
            SI longueur(solution) < plus_petit_nombre_de_pieces ALORS
                plus_petit_nombre_de_pieces ← longueur(solution)
                meilleure_solution ← tableau formé de solution et piece
            FIN SI
        FIN SI
    FIN POUR
    RENVOYER meilleure_solution
FIN FONCTION
```

Cette approche est qualifiée de *top-down* car elle part de l'objectif (la somme à rendre) et descend jusqu'au cas de base (rien à rendre).

Application n°2 :
Utiliser la fonction précédente pour rendre la monnaie sur 20 à partir de pièces de 1 et de 2. Avec les mêmes pièces essayer de rendre 30 puis 33 voire 35. Que remarquez-vous ?
D'où vient le comportement observé ?

2) Les limites de l'algorithme

Dès qu'on a une somme un peu importante à rendre, le nombre d'appel de la fonction augmente de manière explosive. Si on reprend l'exemple précédent avec des pièces de 1 et de 2 en stock, pour rendre 10, la fonction est appelée 232 fois. On passe à 28.656 fois pour un rendu sur 20, 3.524.577 appels pour rendre 30 et 433.494.436 exécutions pour un rendu de 40.
Le problème c'est qu'avec l'appel récursif de la méthode top-down, le même calcul est refait plusieurs fois. Par exemple l'appel `rendu_monnaie_naif(1, [1,2])` est effectué 55, 6765 et 832.040 fois lorsqu'on veut rendre respectivement 10, 20 ou 30.
La complexité de l'algorithme est telle qu'il est clairement inutilisable pour des sommes trop importantes. Il faut donc essayer de trouver une autre façon de faire.

II - La programmation dynamique

1) Origine et principe

Richard Bellman

C'est l'américain Richard Bellman qui au début des années 1950 introduit la programmation dynamique pour résoudre des problèmes de chemins optimaux.

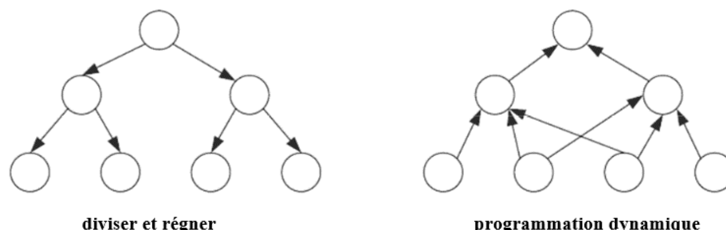
On l'utilise pour des problèmes où on doit trouver la solution optimale dans un nombre fini mais très grand de solutions possibles.

L'idée générale est de construire progressivement la solution du problème en partant de sous-problèmes très simples et en remontant jusqu'à la solution (approche *bottom-up*) en évitant de recalculer plusieurs fois la même chose et éventuellement en décidant de ne pas calculer des sous-problèmes dont on déduit qu'ils n'amèneront pas à la solution optimale.

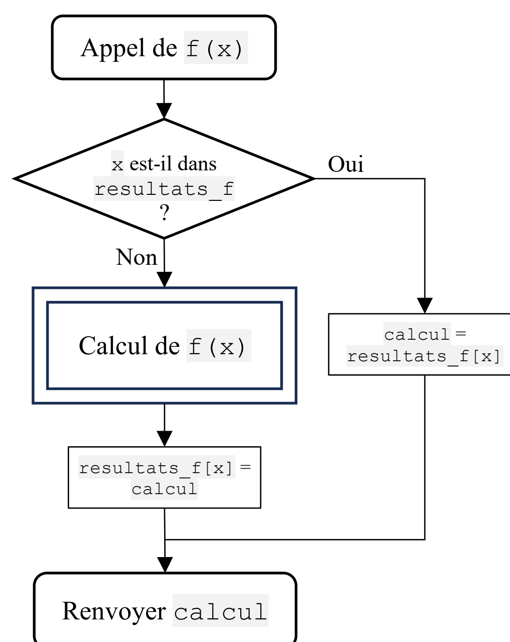
Licence Creative Commons BY-SA

La programmation dynamique est basée sur le principe d'optimalité de Bellman qui stipule que la solution optimale d'un problème est obtenue en combinant les solutions optimales des sous-problèmes dont on peut le déduire.

On n'est pas loin de l'approche de la méthode diviser pour régner du chapitre précédent. La différence essentielle entre les deux vient du fait que dans « diviser pour régner » les sous-problèmes sont indépendants et leurs solutions ne se recoupent pas tandis que dans la programmation dynamique, les sous-problèmes peuvent se superposer. Ainsi la solution d'un sous-problème peut servir à l'élaboration de la solution à plusieurs autres sous-problèmes d'ordres supérieurs.



2) Mémoïsation



La mémoïsation n'est *pas* une technique dc programmation dynamique, mais nous la présentons ici car elle permet d'accélérer grandement et de manière très simple les algorithmes qui appellent plusieurs fois la même fonction avec les mêmes arguments.
Il s'agit simplement de stocker, généralement dans un dictionnaire, les valeurs retournées par la fonction. Avant d'appeler cette dernière, on vérifie simplement si la fonction a déjà été appelée avec les mêmes arguments. Si c'est le cas on récupère le résultat stocké en mémoire sans avoir besoin d'exécuter la fonction. Sinon on appelle la fonction et on stocke sa valeur de retour.
Le stockage et le rappel de la valeur peuvent être faits à l'intérieur de la fonction elle-même ou dans celle qui l'appelle.
La variable qui stocke les résultats doit être conservée entre les appels de la fonction, ce sera donc une variable globale ou du moins englobante.

Application n°3 :
1) Ecrire une fonction `rendu_monnaie_memoise(somme, pieces)` qui utilise la mémoïsation en reprenant le code de la fonction précédente et en ajoutant uniquement les éléments de mémoïsation.
2) Exécutez votre fonction avec les valeurs de l'application précédente. Par quoi est maintenant limité le calcul ?

3) Bottom-up

On cherche maintenant à écrire un algorithme *bottom-up* utilisant la programmation dynamique pour résoudre notre problème de rendu de monnaie.
L'idée est de construire progressivement la solution en rendant d'abord sur 1, puis sur 2, 3, … et ainsi de suite jusqu'à la somme à rendre. A chaque étape on va stocker dans un grand tableau `rendus` un tableau de deux éléments contenant comme premier élément le nombre de pièces à rendre pour arriver à cette somme et comme deuxième élément un tableau contenant les valeurs des pièces à rendre.

Licence Creative Commons BY-SA

Avec les pièces {1, 3, 7} les premières cases du tableau rendus seraient celles représentées ci-contre.

Dans le tableau on ne stockera que la solution optimale.

Dans l'exemple ci-contre, on pourrait rendre 7 avec {1,3,3}, mais cela fait 3 pièces ce qui est plus que la solution optimale de 1 pièce {7}.
Donc à chaque étape on va examiner ce que donnerait l'ajout de chaque pièce disponible et si le résultat est meilleur (moins de pièces), il remplace le résultat précédent.
Il faudra donc dans un premier temps initialiser un tableau `rendus` avec dans chaque case (sauf la 0) un nombre de pièces supérieur au maximum (si on ne rend que des pièces de 1).

Tableau `rendus` pour {1,3,7}	
indice	contenu
0	`[0, []]`
1	`[1, [1]]`
2	`[2, [1, 1]]`
3	`[1, [3]]`
4	`[2, [1, 3]]`
5	`[3, [1, 1, 3]]`
6	`[2, [3, 3]]`
7	`[1, [7]]`
8	`[2, [1, 7]]`
9	`[3, [1, 1, 7]]`
10	`[2, [3, 7]]`
⋮	⋮

Application n°4 :
1) Ecrire une fonction `rendu_monnaie_dynamique(somme, pieces)` qui implémente l'algorithme décrit plus haut.
2) Quelle est la complexité de cet algorithme ?
3) Pour les plus rapides, comparer les temps d'exécution des 3 versions de l'algorithme avec le module `timeit`.

III – Un autre exemple : le plus grand sous-tableau

On cherche quelle est la sous-partie d'un tableau (ensemble de cases contigües du tableau) dont la somme est maximale. On considère bien sûr que le tableau contient des nombres qui peuvent être négatifs.
Par exemple dans le tableau `[4,-1,-6,3,-5,3,4,-3,5,-1]`, le sous-tableau `[3,4,-3,5]` est la plus grande séquence et sa valeur est 9.

1) Algorithme force brute

Pour résoudre le problème du plus grand sous-tableau, on va dans un premier temps utiliser un algorithme de type « force brute » qui essaye toutes les combinaisons possibles de sous-tableau en calculant à chaque fois leurs sommes et qui retient la séquence ayant la plus grande somme.

Application n°5 :
1) Ecrire une fonction `plus_grande_sequence(t)` qui renvoie le sous-tableau dont la somme est maximale dans le tableau `t` en utilisant l'algorithme décrit précédemment.

Par exemple `plus_grande_sequence([2,-11,6,-1,8,-3,2,-13,10])` renvoie `[6, -1, 8]`.

2) Quelle est la complexité de cet algorithme ?

2) Algorithme dynamique

On remarque que ce sont les nombres négatifs qui font qu'il existe des sous-tableaux de sommes plus ou moins importantes car si tous les nombre étaient positifs, le tableau le plus grand serait le tableau complet. Pour autant un seul nombre négatif ne suffit pas à couper une séquence et la plus grande séquence peut contenir des nombres négatifs.
En fait on peut avancer dans le tableau en ajoutant les nombres, mais si à un moment cette somme devient négative, c'est qu'on peut oublier ce début de tableau et qu'il est préférable de recommencer à zéro à partir du moment où la somme devient négative.

Licence Creative Commons BY-SA

On peut illustrer ce fonctionnement avec le tableau ci-contre.

Tableau t	debut	fin	somme	Meilleure somme	Meilleurs indices debut/fin
`[2,-11,6,-1,8,-3,2,-13,10]`	0	0	2	2	0,0
`[2,-11,6,-1,8,-3,2,-13,10]`	0	1	-9	2	0,0
La somme est négative : on repart avec une somme de 0 à l'indice suivant					
`[2,-11,6,-1,8,-3,2,-13,10]`	2	2	6	6	2,2
`[2,-11,6,-1,8,-3,2,-13,10]`	2	3	5	6	2,2
`[2,-11,6,-1,8,-3,2,-13,10]`	2	4	13	13	2,4
`[2,-11,6,-1,8,-3,2,-13,10]`	2	5	10	13	2,4
`[2,-11,6,-1,8,-3,2,-13,10]`	2	6	12	13	2,4
`[2,-11,6,-1,8,-3,2,-13,10]`	2	7	-1	13	2,4
La somme est négative : on repart avec une somme de 0 à l'indice suivant					
`[2,-11,6,-1,8,-3,2,-13,10]`	8	8	10	13	2,4

Application n°6 :

1) Ecrire une fonction `plus_grande_sequence_dynamique(t)` qui renvoie le sous-tableau dont la somme est maximale dans le tableau `t` en utilisant l'algorithme précédant (celui où on repart de zéro lorsque la somme devient négative).
2) Quelle est la complexité de cet algorithme ?

Remarque : On aurait pu également faire un algorithme utilisant le paradigme « diviser pour régner » : on divise le tableau récursivement en 2 et on cherche le plus grand sous-tableau dans la partie de gauche, dans celle de droite puis entre les deux moitiés. Un tel algorithme a une complexité en $O(n \cdot log(n))$ ce qui est donc meilleur que l'algorithme « force brute », mais moins bon que celui en programmation dynamique.

L'essentiel :

....

....

....

....

....

....

....

....

Références :

Programmation dynamique : https://complex-systems-ai.com/algorithmique/programmation-dynamique/

Algorithme de plus grande sous-séquence : http://www.xavierdupre.fr/app/ensae_teaching_cs/helpsphinx/notebooks/exercice_plus_grande_somme.html

Licence Creative Commons BY-SA

TNSI	Algorithmique	Exercices – Programmation dynamique
	Programmation dynamique	

Exercice 1 : Fibonacci

En mathématique la suite de Fibonacci (Mathématicien Italien) est une suite d'entiers définie par :

$$F_0 = 0,\ F_1 = 1 \text{ et } F_n = F_{n-1} + F_{n-2} \text{ pour } n \geq 2$$

1) Programmer une fonction récursive `fibo(n)` qui renvoie le terme de rang n de la suite de Fibonacci.
2) Vérifier la correction de votre fonction avec les valeurs des premiers termes de la suite indiqués ci-contre.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987

3) Essayez votre fonction sur des valeurs de n élevées (≥ 30). Quelle est d'après vous la classe de complexité de votre algorithme ? D'où vient cette complexité explosive ?
4) Ecrire une fonction `fibo_dynamique(n)` qui utilise la programmation dynamique et une approche bottom-up pour calculer le terme de rang `n` de la suite. Quelle est sa complexité ?

Exercice 2 : Exponentiation rapide

L'algorithme RSA de chiffrement asymétrique utilise une fonction d'exponentiation rapide pour calculer x^y modulo n de manière rapide quand x et y sont deux entiers très grands.
L'algorithme vu en cours de sécurisation des communications et reproduit ci-contre permet un calcul rapide pour des entiers x et y de taille modérée.

```
def exponentiation_rapide(a:int, b:int, n:int) -> int:
    """Renvoi a^b [n] calculé par une méthode d'exponentiation rapide"""
    if b == 1:      # Cas de base
        return a % n       # On renvoi a modulo n
    if b % 2 == 0:     # Si l'exposant est pair
        # On renvoit le produit de {a^(b//2) [n]  x a^(b//2) [n]} [n]
        return (  (exponentiation_rapide(a, b//2, n) % n) \
                * (exponentiation_rapide(a, b//2, n) % n) ) % n
    else:  # Si impair
        # On renvoit le produit de {a^(b//2) [n]  x a^(b//2 + 1) [n]} [n]
        return (  (exponentiation_rapide(a, b//2, n) % n)\
                * (exponentiation_rapide(a, b//2 + 1, n) % n) ) % n
```

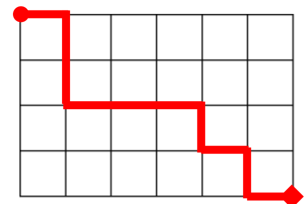
1) L'approche utilisée par l'algorithme est-elle de type *bottom-up* ou *top-down* ?

On peut remarquer qu'un nombre entier peut s'écrire en binaire et qu'on peut donc le décomposer en une somme de puissances de 2. Ainsi $(13)_{10} = (1101)_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1$ et donc $a^{13}\ [n] = a^8 \times a^4 \times a^1$.

Les facteurs de la décomposition (8, 4 et 1 dans l'exemple) sont toujours des puissances de 2 et donc en partant de 1 chaque terme de la décomposition (s'il est présent) peut être déduis en multipliant le précédent par 2.

2) Programmer une version plus efficace de l'exponentiation rapide suivant la programmation dynamique en utilisant la propriété expliquée précédemment. Vérifier que $12^{99999999999}\ [1000] = 448$.

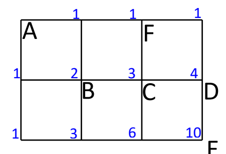
Exercice 3 : Nombre de chemins



On cherche à cherche à connaître le nombre de chemins différents pour aller du coin supérieur gauche d'un quadrillage à son coin inférieur droit.
A chaque étape du déplacement, on a seulement 2 choix :

- se déplacer vers la droite d'une case ;
- se déplacer vers le bas d'une case.

1) Combien y a-t-il de chemins possibles dans un quadrillage de 2×2 cases ? 3×2 cases ?



On remarque qu'en partant du coin supérieur gauche, il n'y a qu'un seul chemin vers les points de bords haut et gauche. Pour aller de A à B, il y a deux chemins (droite-bas ou bas-droite) ; pour aller de A à C, on peut passer par B puis directement à C ou passer par F puis directement C. On a donc 2 chemins par B + 1 par F, donc 3 chemins. De même pour aller à D il y a 3 chemins par C + 1 chemin par F soit 4 possibilités. On remarque ainsi que pour aller au coin inférieur droit d'une case, il y a autant de possibilités que la somme des chemins allant à ses coins bas-gauche et haut-droit. On peut ainsi en partant du point A marquer le nombre de chemins de proches en proches jusqu'à E.

2) En s'inspirant de l'explication précédente, programmer une fonction `chemins(n,m)` qui renvoie le nombre de chemins possibles sur une grille de `n`×`m` cases. Combien y en a-t-il sur une grille de 10×10 ?

Licence Creative Commons BY-SA