Chapitre II
Programmation dynamique

1. Dictionnaires

Cette première partie a pour objet l'étude d'une structure de données que vous avez déjà rencontré en première année : les dictionnaires. Cette structure de données répond à la problématique suivante : comment rechercher efficacement de l'information dans un ensemble géré de façon dynamique (c'est à dire dont le contenu est susceptible d'évoluer au cours du temps).

Les dictionnaires sont couramment utilisées en informatique : c'est par exemple la structure de données utilisée pour gérer les systèmes de noms de domaine (DNS, pour *Domain Name System*). Les ordinateurs connectés à un réseau comme Internet possèdent une adresse numérique (en IPv4 par exemple, celles-ci sont représentées sous la forme xxx.xxx.xxx.xxx, où xxx est un nombre hexadécimal variant entre 0 et 255). Pour faciliter l'accès aux systèmes qui disposent de ces adresses, un mécanisme a été mis en place pour associer un nom (plus facile à retenir) à une adresse IP. Ce mécanisme utilise un dictionnaire dans laquelle les clefs sont les noms de domaine et les valeurs les adresses IP.

1.1 Description

Un *dictionnaire* (ou mieux une *table d'association*) est un type de données associant un ensemble de clefs à un ensemble de valeurs. Plus formellement, si C désigne l'ensemble des clefs et V l'ensemble des valeurs, un dictionnaire est un sous-ensemble T de $C \times V$ tel que pour toute clef $c \in C$ il existe *au plus* un élément $v \in V$ tel que $(c, v) \in T$. Les éléments de T sont appelés des *associations*.

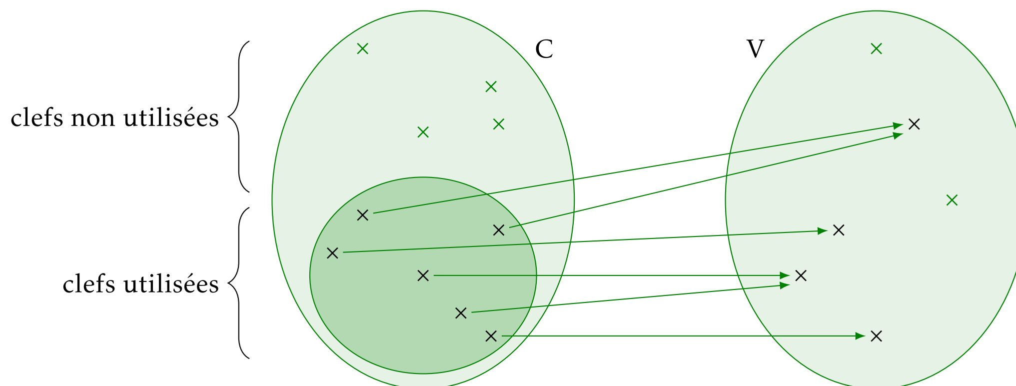


Figure 1 – Représentation informelle d'un dictionnaire.

Un dictionnaire supporte en général les opérations suivantes :

- *ajout* d'une nouvelle association $(c, v) \in C \times V$ dans T ;
- *suppression* d'une association (c, v) de T ;
- *existence* d'une association (c, v) dans T pour une clef $c \in C$ donnée ;
- *lecture* de la valeur v associée à une clef c présente dans T.

En Python, la création d'un dictionnaire se réalise en suivant la syntaxe `{c1: v1, ... , cn: vn}` où $c_1, \ldots, c_n$ sont des clefs (nécessairement deux-à-deux distinctes) et $v_1, \ldots, v_n$ les valeurs qui leur sont associées. Ainsi, `{ }` crée un dictionnaire vide.

Si `D` est un dictionnaire et `c` une clef,

- `D[c] = v` ajoute une nouvelle association si la clef n'est pas présente dans le dictionnaire, et modifie l'association précédente sinon ;

– `del D[c]` supprime une association si la clef est présente dans le dictionnaire, et déclenche l'exception `KeyError` sinon ;
– l'expression `c in D` renvoie un booléen indiquant si la clef est présente ou non dans le dictionnaire ;
– `D[c]` renvoie la valeur associée à la clef si celle-ci est présente dans le dictionnaire, et déclenche l'exception `KeyError` sinon.

Notons que, pour des raisons qui apparaîtrons plus loin, les clefs d'un même dictionnaire doivent impérativement appartenir à un type immuable.

■ Parcours d'un dictionnaire

Dans certaines situations, il peut être utile de parcourir l'ensemble des clefs d'un dictionnaire, l'ensemble des valeurs d'un dictionnaire, voire les deux. C'est pourquoi trois méthodes sont associées aux objets de ce type :

– `D.keys` renvoie la liste[1] des clefs d'un dictionnaire `D` ;
– `D.values` renvoie la liste des valeurs présentes dans le dictionnaire `D` ;
– `D.items` renvoie la liste des couples (clefs, valeurs) du dictionnaire `D`.

Ainsi, pour parcourir un dictionnaire on utilisera suivant les besoins l'une de ces trois syntaxes :

```
for k in D.keys:     for v in D.values:     for (k, v) in D.items:
```

Notons que l'itération sur les clefs `for k in D.keys:` peut même s'écrire plus simplement `for k in D:`

1.2 Mise en œuvre pratique d'un dictionnaire

Il est communément admis (même si ce n'est pas tout-à-fait correct) que les quatre opérations de base d'un dictionnaire se réalisent avec une complexité temporelle constante ; pour comprendre comment une telle performance est possible, nous allons décrire quelques méthodes d'implémentation de cette structure.

■ Deux solutions inenvisageables en pratique

Si les clefs étaient des entiers compris entre 0 et $m-1$, le problème serait simple : il suffirait d'utiliser un tableau de taille m. Comme ce n'est en général pas le cas, on se ramène à cette situation en utilisant une fonction $f : \mathrm{C} \to [\![0, m-1]\!]$ associant aux différentes clefs $c \in \mathrm{C}$ possibles un entier dans $[\![0, m-1]\!]$.
Mais un autre problème, bien plus grave, se pose : l'ensemble C est en général de cardinal extrêmement grand : ce peut-être par exemple l'ensemble des objets de type `int`, ou `float`, donc un cardinal de l'ordre de 2^{64}, et l'espace mémoire dévolu à un dictionnaire serait bien plus grand que nos moyens de stockages actuels (il faudrait plus de 100 milliards de tera-octets pour stocker un tel tableau...)

Une autre solution consisterait à stocker un dictionnaire dans une liste dynamique, initialement vide, que l'on remplirait progressivement avec des couples clef/valeur. Mais ici se pose un problème de performance, car les opérations usuelles d'un dictionnaire : lecture, ajout, suppression seraient de complexité linéaire, alors qu'on s'attend à une complexité constante (ou presque).

```python
def find_lst(L, clef):
    for (c, v) in L:
        if c == clef:
            return v
    raise KeyError(clef)
```

```python
def add_lst(L, clef, valeur):
    for k in range(len(L)):
        (c, v) = L[k]
        if c == clef:
            L[k] = (clef, valeur)
            return None
    L.append((clef, valeur))
```

FIGURE 2 – Les opérations de lecture et d'ajout dans une liste sont de complexité linéaire.

1. En toute rigueur ce n'est pas une liste mais vous pouvez faire comme si.

■ Une solution médiane

La solution consiste à adopter une solution qui mélange les deux approches précédentes : nous allons utiliser un tableau de taille m et une fonction $f : \mathrm{C} \to [\![0, m-1]\!]$ qui à toute clef associe un entier, mais avec une valeur m bien plus petite que le cardinal de C. Une telle fonction ne sera donc pas injective : il existera des couples (c_1, c_2) dans C tels que $c_1 \neq c_2$ et $f(c_1) = f(c_2)$. Un tel couple (c_1, c_2) est appelé une *collision*.
Ainsi, deux clefs qui entrent en collision vont être associées à la même case du tableau, et c'est pourquoi chaque case du tableau contiendra une liste dans laquelle les associations clefs/valeurs seront stockées suivant le principe décrit plus haut.
La figure 3 illustre cette solution avec $f(c_1) = f(c_4)$ et $f(c_3) = f(c_5) = f(c_6)$.

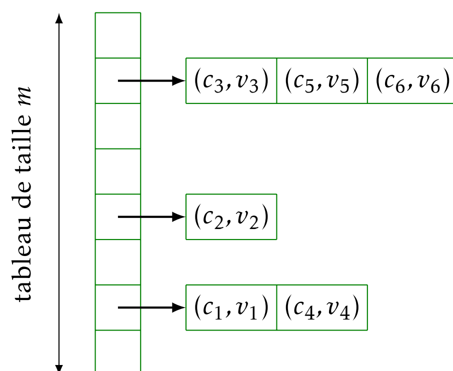


FIGURE 3 – Les clefs qui entrent en collision sont stockées dans le même paquet.

```
def find(D, clef):
    L = D[f(clef)]
    return find_lst(L, clef)
```

```
def add(D, clef, valeur):
    L = D[f(clef)]
    add_lst(L, clef, valeur)
```

FIGURE 4 – Les opérations de lecture et d'ajout dans un dictionnaire.

1.3 Un peu de probabilités

L'idéal consiste, on s'en doute, à remplir le dictionnaire en faisant en sorte que les m listes soient de tailles à peu près égales. Dans ce cas, le temps d'exécution par rapport à la solution naïve utilisant une seule liste serait divisé par m, ce qui constituerait une amélioration loin d'être négligeable.

Formalisons un peu ceci : on considère l'espace probabilisé $(\Omega, \mathcal{A}, \mathbb{P})$ où $\Omega = \mathrm{C}$ est l'ensemble des clefs, $\mathcal{A} = \mathscr{P}(\Omega)$ et $\mathbb{P}$ la probabilité définie sur $\mathcal{A}$ par $\mathbb{P}(\{c\}) = \dfrac{1}{\operatorname{card} \mathrm{C}}$.
Faisons ensuite l'hypothèse (importante) que $\mathrm{F} : c \mapsto f(c)$ est une *variable aléatoire suivant une loi uniforme*, et intéressons-nous au problème de la recherche d'une clef dans un dictionnaire contenant n enregistrements.
Ces n enregistrements sont modélisés par une suite $(\mathrm{F}_1, \dots, \mathrm{F}_n)$ de variables aléatoires indépendantes et identiquement distribuées, de même loi que F, et la clef recherchée dans le dictionnaire est représentée par F_{n+1}.
On note $\mathrm{N} = \operatorname{card}\left\{k \in [\![1, n]\!] \;\middle|\; \mathrm{F}_k = \mathrm{F}_{n+1}\right\}$ le nombre de collisions entre F_{n+1} et les clefs déjà présentes dans le dictionnaire et X le nombre de comparaisons nécessaires pour trouver cette clef.
On admettra que :

- *si la clef recherchée ne se trouve pas dans le dictionnaire, alors* $\mathbb{E}(\mathrm{X}) = \dfrac{n}{m}$;
- *si la clef recherchée se trouve dans le dictionnaire, alors* $\mathbb{E}(\mathrm{X}) = \dfrac{n-1}{2m} + 1 \approx \dfrac{n}{2m}$.

■ Complexité dans le pire des cas et complexité en moyenne

Nous venons de calculer ce qu'il convient d'appeler la *complexité en moyenne* de la recherche d'une clef dans un dictionnaire : le gain en temps est en moyenne amélioré d'un facteur $1/m$, et lorsque m est du même ordre de grandeur que n, les espérances calculées sont bornées et il est alors légitime de parler de *complexité constante en moyenne*.

Attention cependant à ne pas confondre cette complexité avec la complexité dans le pire des cas, qui correspond ici à la situation (très improbable) où toutes les clefs utilisées seraient stockées dans la même liste et où la recherche nécessiterait dans le pire des cas à n comparaisons. Mais à cet égard, l'inégalité de Bienaymé-Tchebichev peut nous rassurer : les listes seront courtes, sauf dans des cas extrêmement rares.

Remarque. La notion de complexité en moyenne n'étant pas au programme, on considérera par la suite que les complexités relatives aux fonctions de base d'un dictionnaire sont constantes lorsqu'il s'agira de faire l'analyse de la complexité temporelle d'un algorithme.

1.4 Fonction de hachage

Parlons maintenant un peu de la fonction f : pour obtenir de bonnes performances nous avons dit qu'il fallait que cette fonction réalise deux objectifs :

– être facile à calculer ;

– avoir une distribution la plus uniforme possible.

Pour définir cette fonction, on utilise une fonction h, appelée *fonction de hachage*, associant un entier à une clef de C, et on définit f en posant :

$$f(c) = h(c) \bmod m$$

Choix de la fonction de hachage

C'est un problème complexe que nous n'aborderons pas. Sachez seulement que Python propose une fonction de hachage : si `x` est un objet immuable (`int`, `str`, `float`, ...) alors `hash(x)` renvoie un entier répondant autant que faire ce peut aux exigences d'une fonction de hachage :

```
In [1]: hash(12345)
Out[1]: 12345

In [2]: hash("12345")
Out[2]: -3486454717630806608

In [3]: hash((1, 2, 3, 4, 5))  # un tuple est immuable
Out[3]: -5659871693760987716

In [4]: hash([1, 2, 3, 4, 5])  # une liste est mutable
TypeError: unhashable type: 'list'
```

Une fonction de hachage ne peut prendre en entrée une variable mutable, car elle doit être déterministe : le résultat d'un hachage doit être toujours le même sur la même entrée (or le contenu d'une variable mutable peut être modifié sans modifier la dite variable...)

Remarque. Une fonction de hachage a d'autres usages. Par exemple, lorsque vous choisissez un mot de passe sur un site sécurisé, ce dernier ne va pas stocker votre mot de passe en clair, mais va stocker le résultat de l'application d'une fonction de hachage h à votre mot de passe. Sachant qu'il est très difficile de trouver les antécédents d'un entier par la fonction h, pirater le site ne mettra pas en cause la sécurité de votre mot de passe. Une autre application des fonctions de hachage est le contrôle d'intégrité d'un fichier informatique : à chaque fichier est associé une valeur par une fonction de hachage (vous connaissez peut-être le MD5) qui permet de vérifier si un fichier téléchargé a été transmis correctement.

2. Programmation dynamique

2.1 Un problème posé par la programmation récursive

Nous allons nous intéresser au calcul du coefficient binomial $\binom{n}{p}$. Une solution consiste à utiliser la programmation récursive et la formule de Pascal, ce qui nous amène à écrire :

```
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n - 1, p - 1) + binom(n - 1, p)
```

Malheureusement, cette fonction s'avère très peu efficace, même pour de relativement faibles valeurs de n et p : à titre d'illustration, il faut 54 secondes à mon ordinateur pour calculer $\binom{30}{15}$.

La raison en est facile à comprendre : lorsqu'on observe par exemple l'arbre de calcul de $\binom{5}{2}$ on constate que des appels récursifs sont identiques et donc superflus (figure 5).

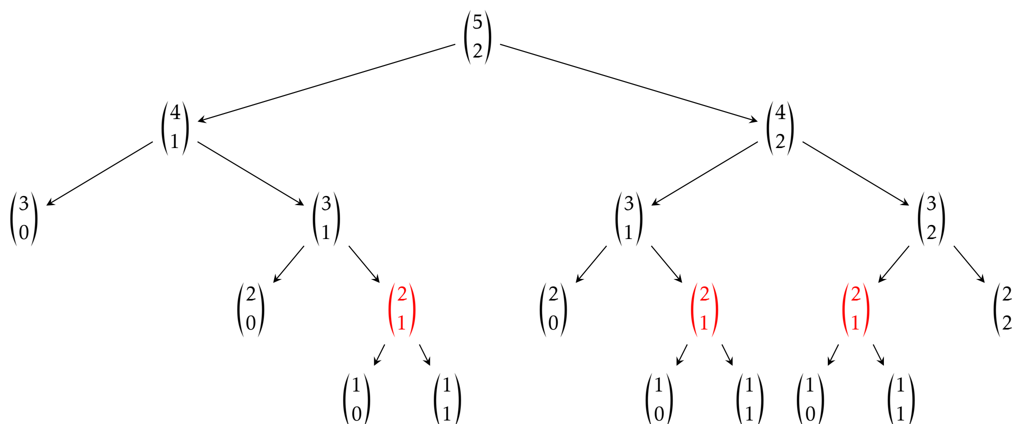


Figure 5 – Le calcul de $\binom{5}{2}$ fait appel trois fois au calcul de $\binom{2}{1}$.

Nous pouvons par exemple constater que le calcul de $\binom{5}{2}$ nécessite de calculer trois fois $\binom{2}{1}$. Et l'expérience montre que le calcul de $\binom{30}{15}$ fait appel 40 116 600 fois (!) au calcul de $\binom{2}{1}$.

Complexité temporelle
Pour évaluer la complexité de cette fonction, on note $\mathrm{C}(n,p)$ le nombre d'additions réalisées par cette fonction, on dispose des relations :

$$\mathrm{C}(n,0) = \mathrm{C}(n,p) = 0 \quad \text{et} \quad \forall p \in [\![1, n-1]\!], \quad \mathrm{C}(n,p) = \mathrm{C}(n-1,p-1) + \mathrm{C}(n-1,p) + 1$$

On démontre alors par récurrence sur $n \in \mathbb{N}$ que pour tout $p \in [\![0,n]\!]$, $\mathrm{C}(n,p) = \binom{n}{p} - 1$. Or la formule de Stirling permet d'établir l'équivalent : $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$; le calcul de $\binom{2n}{n}$ par cette fonction est donc de complexité exponentielle.

Où est le problème ?

Le problème à résoudre, ici le calcul de $\binom{n}{p}$, se ramène à la résolution de deux sous-problèmes : le calcul de $\binom{n-1}{p-1}$ et de $\binom{n-1}{p}$, *sous-problèmes qui sont en interaction.*

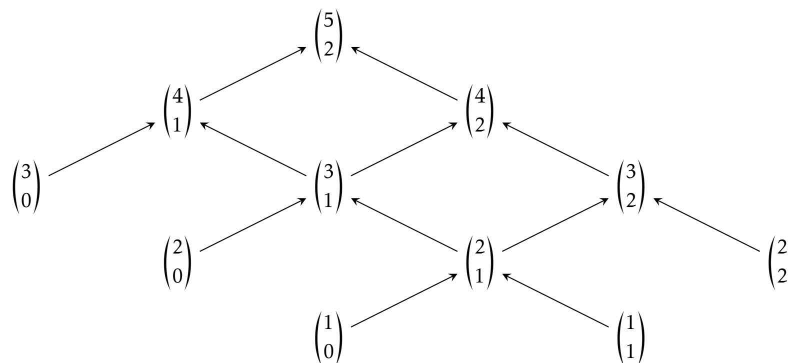
Par exemple, on constate sur la figure 5 que le calcul de $\binom{4}{1}$ et le calcul de $\binom{4}{2}$ font tous deux appel au même sous-problème : le calcul de $\binom{3}{1}$.

Ainsi, la présence de sous-problèmes en interaction peut faire croitre très rapidement la complexité d'une fonction, au point d'en rendre son usage rédhibitoire.

■ La solution proposée par la programmation dynamique

La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands.

Concrètement, le calcul de $\binom{5}{2}$ se réalise en suivant le schéma :



Pour réaliser ce type de solution on utilise souvent un tableau, ici un tableau bi-dimensionnel $(n+1)\times(p+1)$ (dont seule la partie pour laquelle $i \geqslant j$ sera utilisée). Ce tableau sera progressivement rempli par les valeurs des coefficients binomiaux, en commençant par les plus petits (figure 6).

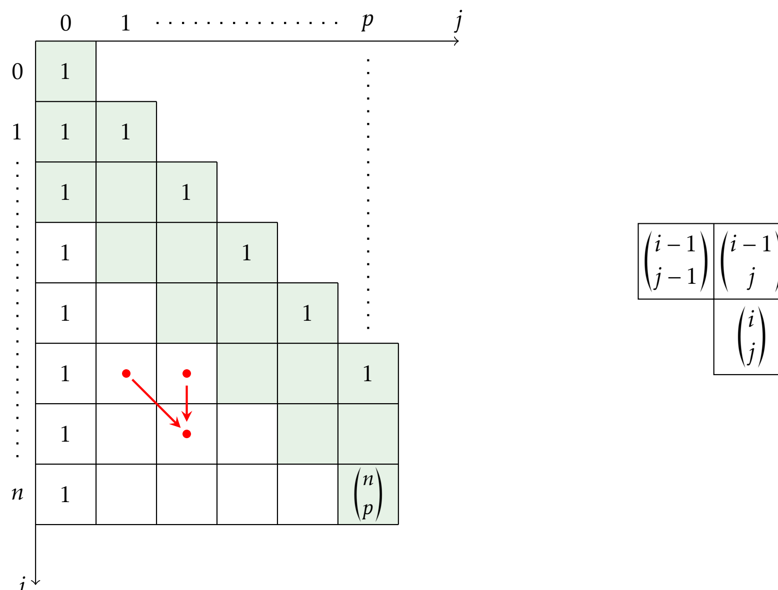


Figure 6 – Le schéma de dépendance du calcul de $\binom{n}{p}$.

Il faut faire attention à bien respecter la relation de dépendance (modélisée par les flèches sur le schéma ci-dessus) pour remplir les cases de ce tableau : la case destinée à recevoir la valeur de $\binom{i}{j}$ ne peut être remplie qu'après les cases destinées à recevoir $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$.

```
1  def binom(n, p):
2      t = [[0 for j in range(p + 1)] for i in range(n + 1)]
3      for i in range(0, n + 1):
4          t[i][0] = 1
5      for i in range(1, p + 1):
6          t[i][i] = 1
7      for i in range(2, n + 1):
8          for j in range(1, min(p, i) + 1):
9              t[i][j] = t[i - 1][j - 1] + t[i - 1][j]
10     return t[n][p]
```

Au prix d'un coût spatial (la création du tableau) cet algorithme est bien plus efficient que l'algorithme récursif initial puisque sa complexité temporelle et spatiale est maintenant en $O(np)$.

Remarque. Notons que cette solution n'est pas encore optimale : il est facile de constater sur le schéma de dépendance que l'algorithme ci-dessus remplit des cases inutiles pour le calcul de $\binom{n}{p}$: seules celles qui sont colorées sont nécessaires. On peut d'ailleurs observer qu'on peut se contenter d'utiliser un tableau unidimensionnel contenant les valeurs $\binom{i+j}{j}$ pour $j \in [\![0, p]\!]$ et de faire varier i entre 0 et $n-p$:

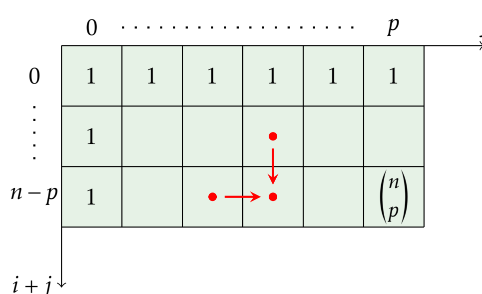


FIGURE 7 – Le schéma de dépendance du calcul de $\binom{i+j}{j}$.

```
def binom(n, p):
    t = [1 for j in range(p + 1)]
    for i in range(n - p):
        for j in range(1, p + 1):
            t[j] = t[j] + t[j - 1]
    return t[p]
```

La complexité est maintenant un $O(p(n-p))$.

■ Mémoïsation

Un inconvénient de la programmation dynamique réside dans la perte de lisibilité de l'algorithme, comparativement à l'algorithme récursif. L'idéal serait donc de combiner l'élégance de la programmation récursive avec l'efficacité de la programmation dynamique.

La solution existe, elle porte le nom de *mémoïsation*. Elle consiste à associer à la fonction un dictionnaire qui va mémoriser le résultat du calcul réalisé. Ainsi, à chaque fois que le programme aura besoin de calculer une valeur, il ira voir dans le dictionnaire si la valeur dont il a besoin a déjà été calculée, et ne réalisera le calcul que dans le cas contraire, en ajoutant ensuite la nouvelle valeur calculée au dictionnaire.

Le calcul du coefficient binomial va alors prendre la forme qui suit :

```
1  binom_dict = {}
2
3  def binom(n, p):
4      if (n, p) not in binom_dict:
5          if p == 0 or n == p:
6              b = 1
7          else:
8              b = binom(n - 1, p - 1) + binom(n - 1, p)
9          binom_dict[(n, p)] = b
10     return binom_dict[(n, p)]
```

On peut observer que le programme récursif se retrouve presque mot pour mot lignes 5 à 8.
Calculons $\binom{5}{2}$ avec cette fonction, puis observons le contenu du dictionnaire :

```
In [1]: binom(5, 2)
Out[1]: 10

In [2]: binom_dict
Out[2]: {(3, 0): 1, (2, 0): 1, (1, 0): 1, (1, 1): 1, (2, 1): 2, (3, 1): 3, (4, 1): 4,
         (2, 2): 1, (3, 2): 3, (4, 2): 6, (5, 2): 10}
```

On peut constater qu'on y retrouve les 10 valeurs nécessaires pour réaliser ce calcul. J'ai représenté figure 8 l'ordre dans lequel ces valeurs ont été introduites dans le dictionnaire.

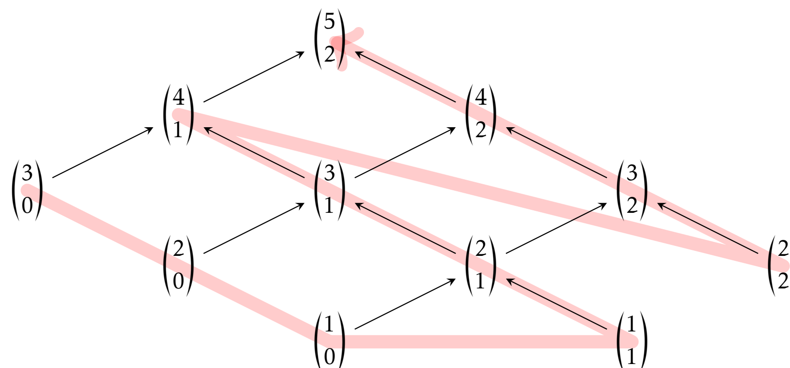


Figure 8 – Ordre d'entrée dans le dictionnaire.

2.2 Programmation dynamique et gloutonne

Tout comme les problèmes que l'on résout par une méthode « diviser pour régner », les problèmes que l'on résout par la programmation dynamique se ramènent à la résolution de sous-problèmes de tailles inférieures. Mais à la différence de la méthode « diviser pour régner », ces sous-problèmes *ne sont pas indépendants*, ce qui impose d'accompagner la programmation récursive par une analyse fine des relations de dépendance, ou beaucoup plus simplement par l'utilisation de la mémoïsation qui gère les relations de dépendance à notre place.

■ Problèmes d'optimisation

La programmation dynamique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique dès lors que la solution optimale peut être déduite des solutions optimales des sous-problèmes (c'est le *principe d'optimalité* de Bellman, du nom de son concepteur). Cette méthode garantit d'obtenir la meilleure

programmation dynamique *programmation gloutonne*

Figure 9 – Une illustration des programmations dynamique et gloutonne.

solution au problème étudié, mais dans un certain nombre de cas sa complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.
Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la *programmation gloutonne*. Alors que la programmation dynamique se caractérise par la résolution par taille croissante de *tous* les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un problème local *et un seul* en suivant une heuristique (c'est à dire une stratégie permettant de faire un choix rapide mais pas nécessairement optimal). On ne peut en général garantir que la stratégie gloutonne détermine la solution optimale, mais lorsque l'heuristique est bien choisie on peut espérer obtenir une solution proche de celle-ci.

Pour apprécier la différence entre programmation dynamique et programmation gloutonne, nous allons maintenant étudier le problème suivant :

Exercice 1

En partant du sommet du triangle ci-dessous et en se déplaçant vers les nombres adjacents de la ligne inférieure, le total maximum que l'on peut obtenir pour relier le sommet à la base est égal à 28 :

```
        3
      7   4
    2   4   6
  8   5   9   3
6   8   4   3   1
```

$3+7+2+8+8=28$

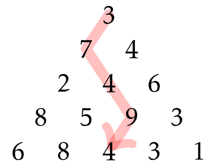
```
t = [[3],
     [7, 4],
     [2, 4, 6],
     [8, 5, 9, 3],
     [6, 8, 4, 3, 1]]
```

Rédiger une fonction calculant le total maximum d'un chemin reliant le sommet à la base d'un tel triangle de hauteur n. On pourra considérer que les valeurs de ce triangle sont stockées dans un tableau bi-dimensionnel $n \times n$ (autrement dit, `t[i][j]` contient la $(j+1)^{\text{e}}$ valeur de la $(i+1)^{\text{e}}$ ligne).

On peut commencer par rédiger une solution gloutonne : elle consiste à chaque étape à choisir entre le déplacement de gauche ou le déplacement de droite en se dirigeant vers la valeur la plus grande :

```
def glouton(t):
    j = 0
    s = t[0][0]
    for i in range(len(t) - 1):
        if t[i + 1][j] < t[i + 1][j + 1]:
            s += t[i + 1][j + 1]
            j += 1
        else:
            s += t[i + 1][j]
    return s
```

Mais cette solution ne donne pas la solution optimale, comme le montre l'exemple introductif :

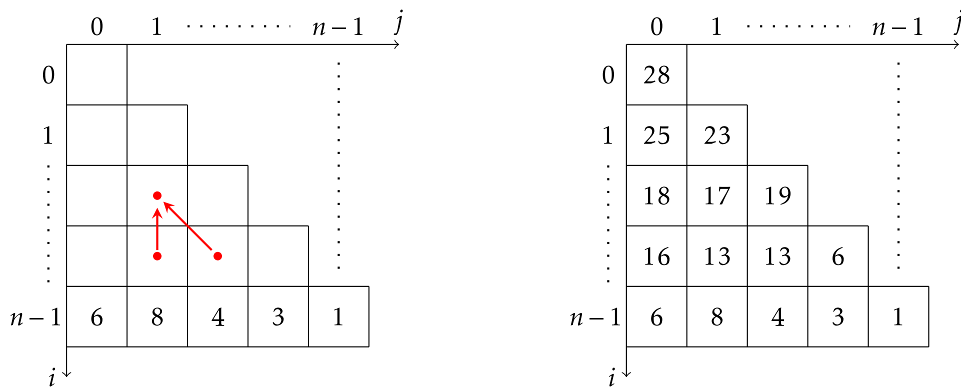


$$3+7+4+9+4=27$$

La solution dynamique consiste à noter $f(i,j)$ la valeur maximale que l'on peut obtenir en partant de la case (i,j). Le but est alors de calculer $f(0,0)$.
Il est facile de se convaincre que pour tout $j \in [\![0,n-1]\!]$, $f(n-1,j) = t[n-1][j]$ et que pour tout $i \in [\![0,n-2]\!]$, pour tout $j \in [\![0,i]\!]$,

$$f(i,j) = t[i][j] + \max\big(f(i+1,j), f(i+1,j+1)\big)$$

Le problème revient donc à compléter le tableau suivant en suivant la règle de remplissage indiquée par la relation ci-dessus :




```
def dynamique(t):
    n = len(t)
    f = [[0 for j in range(i + 1)] for i in range(n)]
    for j in range(n):
        f[n - 1][j] = t[n - 1][j]
    for i in range(n - 2, -1, -1):
        for j in range(i + 1):
            f[i][j] = t[i][j] + max(f[i + 1][j], f[i + 1][j + 1])
    return f[0][0]
```

Enfin, se pose le problème de déterminer non seulement la valeur maximale mais aussi le chemin qui la réalise. Pour ce faire, on modifie la dernière ligne de la fonction `dynamique` pour renvoyer le tableau f plutôt que la valeur $f[0][0]$: on remplace `return f[0][0]` par `return f`.
Il reste alors à descendre dans ce tableau en choisissant à chaque étape, entre les deux valeurs inférieures, celle qui est la plus grande :

```
def cheminOptimal(t):
    f = dynamique(t)
    j = 0
    chemin = ''
    for i in range(len(t) - 1):
        if f[i + 1][j] < f[i + 1][j + 1]:
            j += 1
            chemin += 'D'
        else:
            chemin += 'G'
    return chemin
```

2.3 Le problème du sac à dos

Il se pose dans les termes suivants :

> *étant donnés n objets de valeurs $c_1, \dots, c_n$ et de poids respectifs $w_1, \dots, w_n$, comment remplir un sac à dos maximisant la valeur emportée $\sum_{i \in I} c_i$ tout en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{max}$?*

Pour élaborer un algorithme glouton résolvant le problème, il faut définir une heuristique, ici un critère de priorité pour le choix des objets à prendre. Nous pouvons par exemple choisir en priorité les objets dont le rapport $\frac{\text{valeur}}{\text{poids}} = \frac{c_i}{w_i}$ est maximal, et remplir le sac tant que c'est possible :

```python
def glouton(c, w, Wmax):
    r = sorted(
        [(c[k], w[k]) for k in range(len(c))], key=lambda t: t[0] / t[1], reverse=True
    )
    s, p = 0, Wmax
    for k in range(len(c)):
        if r[k][1] <= p:
            s += r[k][0]
            p -= r[k][1]
    return s
```

Pour évaluer la qualité de cette heuristique, j'ai réalisé 1 000 expériences pour chacune desquelles j'ai :

- pris au hasard 100 objets de valeurs comprises entre 1 et 20 et de poids compris entre 1 et 30 ;
- calculé la valeur optimale obtenue pour un poids maximal égal à 300 à la fois par l'algorithme glouton ci-dessus et par l'algorithme dynamique que nous étudierons plus loin.

Sur les 1 000 expériences, 421 ont donné le même résultat pour chacun des deux algorithmes, et dans le cas des 579 autres, l'algorithme glouton a toujours rendu un résultat au moins égal à 98% du résultat de l'algorithme dynamique.
On peut donc considérer ici qu'à défaut de donner un résultat toujours exact, l'algorithme glouton donne un résultat acceptable tout en ayant une complexité moindre que l'algorithme dynamique.

La solution dynamique

Pour résoudre ce problème, nous allons noter $f(k, W)$ la valeur maximale qu'il est possible d'atteindre avec les k premiers objets pour un poids total égal à W.
Si l'objet d'indice k est dans la solution optimale, alors $w_k \leqslant W$ et $f(k, W) = c_k + f(k-1, W - w_k)$; s'il n'y est pas alors $f(k, W) = f(k-1, W)$. On en déduit :

$$f(k, W) = \begin{cases} \max\big(c_k + f(k-1, W - w_k), f(k-1, W)\big) & \text{si } w_k \leqslant W \\ f(k-1, W) & \text{sinon} \end{cases}$$

Pour calculer cette valeur, nous allons utiliser un tableau bi-dimensionnel de taille $(n+1) \times (W_{max}+1)$ destiné à contenir les valeurs de $f(k, w)$ pour $k \in [\![0, n]\!]$ et $W \in [\![0, W_{max}]\!]$.
Nous prendrons comme valeurs initiales $f(0, W) = f(k, 0) = 0$, et notre but est de calculer $f(n, W_{max})$.

Pour remplir ce tableau, il est primordial de respecter l'ordre de dépendance des cases de ce tableau : la case $f(k, W)$ ne peut être calculée que lorsque les cases $f(k-1, W)$ et $f(k-1, W - w_k)$ auront été remplies.
En considérant que les valeurs c_k et w_k sont données sous forme de tableaux, on en déduit l'algorithme :

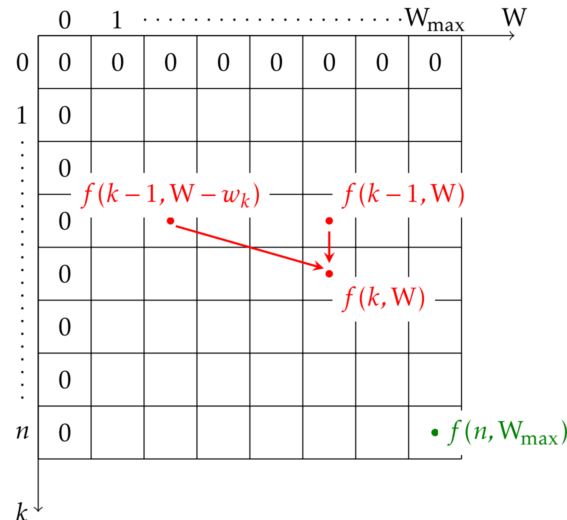


FIGURE 10 – Ordre de dépendance du sac à dos.

```
def sacAdos(c, w, Wmax):
    n = len(c)
    f = [[0 for j in range(Wmax + 1)] for i in range(n + 1)]
    for k in range(n):
        for W in range(Wmax + 1):
            if w[k] <= W:
                f[k + 1][W] = max(c[k] + f[k][W - w[k]], f[k][W])
            else:
                f[k + 1][W] = f[k][W]
    return f[n, Wmax]
```

Il apparaît clairement que la complexité temporelle de cet algorithme est proportionnel au produit nW_{max}, soit en $O(nW_{max})$. L'algorithme glouton quant à lui est en $O(n \log n)$ (le coût du tri).

Remarque. Nous n'avons pas utilisé ici la technique de mémoïsation pour résoudre le problème. Cette dernière, lorsqu'elle est utilisée, nous permet de moins nous préoccuper de l'ordre de dépendance qui est géré par la récursivité :

```
def sacAdos(c, w, Wmax):
    dico = {}
    def f(k, W):
        if (k, W) not in dico:
            if k == 0 or W == 0:
                x = 0
            elif w[k - 1] <= W:
                x = max(c[k - 1] + f(k - 1, W - w[k - 1]), f(k - 1, W))
            else:
                x = f(k - 1, W)
            dico[(k, W)] = x
        return dico[(k, W)]
    return f(len(c), Wmax)
```

Remarque. Cet algorithme calcule la valeur maximale qui peut être emportée dans le sac, mais pas la façon d'y parvenir. Pour la connaître il faut utiliser le tableau (ou le dictionnaire) calculé par la fonction précédente, et retrouver le chemin qui mène de la case initiale à la case finale.

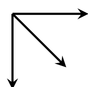
Par exemple, si on modifie la fonction non récursive (la première) pour qu'elle renvoie le tableau `f` qui a été calculé au lieu de la valeur `f[len(c)][Wmax]`, le fonction qui détermine les objets à choisir s'écrira :

```python
def objetsAchoisir(c, w, Wmax):
    f = sacAdos(c, w, Wmax)
    sac = []
    k, W = len(c), Wmax
    while k > 0:
        if f[k][W] > f[k - 1][W]:
            sac.append((c[k - 1], w[k - 1]))
            W -= w[k - 1]
        k -= 1
    return sac
```

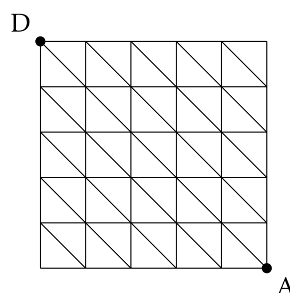
3. Exercices

Exercice 2

Partant du coin supérieur gauche d'une grille $n \times n$, on souhaite calculer le nombre de chemins menant au coin inférieur droit en ne suivant que les trois directions suivantes :



les trois directions possibles



Rédiger une fonction `chemins(n)` qui répond à la question.

Exercice 3

Un *système monétaire* est représenté par la liste des valeurs faciales de ces pièces et billets. Par exemple (en faisant abstraction des centimes), le système monétaire européen est représenté par la liste `s = [1,2,5,10,50,100,200,500]`. On supposera systématiquement que `s[0] = 1`.
Rédiger une fonction `minimum(s, n)` qui prend pour arguments un système monétaire s et un entier n et qui renvoie le nombre minimal de pièces ou de billets nécessaire pour atteindre la valeur n.

Exercice 4

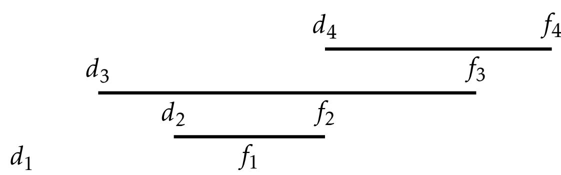
Étant donné une chaîne de caractères $a = a_1 a_2 \cdots a_m$, on appelle *sous-chaîne de longueur k* toute chaîne de caractère $a_{i_1} a_{i_2} \cdots a_{i_k}$ avec $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant m$.

a. Rédiger une fonction `pgscc(a, b)` qui, étant donné deux chaînes de caractères $a = a_1 \cdots a_m$ et $b = b_1 \cdots b_n$, renvoie la longueur de la plus grande sous-chaîne commune à a et b.

b. Modifier votre algorithme pour qu'il renvoie cette fois une sous-chaîne commune de longueur maximale.

Exercice 5 [Ordonnancement d'intervalles pondérés]

On considère une succession d'intervalles de temps $[d_i, f_i]$, $1 \leqslant i \leqslant n$, chacun d'eux étant associé à une valeur v_i. On dit que deux intervalles $[d_i, f_i]$ et $[d_j, f_j]$ *ne se chevauchent pas* lorsque $f_i \leqslant d_j$ ou $f_j \leqslant d_i$. Rédiger une fonction `ordonnancement(d, f, v)` qui détermine la somme maximale des valeurs d'une sous-famille d'intervalles ne se chevauchant pas. On supposera $f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$.



Exercice 6

Si A et B sont deux matrices de tailles respectives $a \times b$ et $c \times d$, le produit AB n'est possible que si $b = c$, et dans ce cas la matrice produit AB est de taille $a \times d$, et peut être calculée à l'aide de acd multiplications.

Sachant que le produit matriciel est associatif mais pas commutatif, le produit ABC peut être calculé de deux manières : (AB)C ou A(BC), qui ne nécessitent pas *a priori* le même nombre de multiplications. Par exemple, si A est de taille 10×100, B de taille 100×5, et C de taille 5×50, le produit (AB)C nécessite $5000 + 2500 = 7500$ multiplications et le produit A(BC), $25000 + 50000 = 75000$ multiplications.

On considère une chaîne de matrices $M_0 M_1 M_2 \cdots M_{n-1}$ de tailles respectives $m_0 \times m_1$, $m_1 \times m_2$, ..., $m_{n-1} \times m_n$. Rédiger une fonction calculant le nombre minimal de multiplications nécessaires pour effectuer ce produit matriciel. On pourra considérer que les valeurs de $m_0, m_1, \ldots, m_n$ sont rangées dans un tableau de taille $n + 1$.