ISM MP, Informatique
Année 2026/2027

Corrigé du TP Informatique 8

Exercice 1

1. On saisit :

```
def fibo1(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibo1(n-1)+fibo1(n-2)
```

2. On saisit :

```
def fibo2(n):
    memo=[None]*(n+1)
    def fibo_memo(k):
        # fonction locale, memo vue comme globale ici
        if memo[k]==None:
            if k==0:
                memo[0]=0
            elif k==1:
                memo[1]=1
            else:
                memo[k]=fibo_memo(k-1)+fibo_memo(k-2)
        return memo[k]
    return fibo_memo(n)
```

ou aussi, avec un dictionnaire et sans mémoïser les cas de base :

```
def fibo2(n):
    memo={}
    def fibo_memo(k):
        # fonction locale, memo vue comme globale ici
        if k==0:
            return 0
        elif k==1:
            return 1
        else:
            if k not in memo:
                res=fibo_memo(k-1)+fibo_memo(k-2)
                memo[k]=res
        return memo[k]
    return fibo_memo(n)
```

3. On saisit :

```
def fibo3(n):
    res=[0,1]
    for k in range(n):
        res.append(res[k]+res[k+1])
    return res[n]
```

4. La fonction `fibo1` trouve rapidement ses limites : elle de complexité temporelle en $O(\varphi^n)$ avec $\varphi > 1$. La fonction `fibo2` est bien meilleure mais demeure contrainte par la hauteur de pile de récursions. La fonction `fibo3` s'affranchit de cette limitation et s'avère plus rapide du fait de la non gestion des empilements de récursions.

Exercice 2

1. On saisit :

```
def cata1(n):
    if n==0:
        return 1
    else:
        s=0
        for i in range(n):
            s+=(cata1(i)*cata1(n-1-i))
        return s
```

2. On saisit :

```
def cata2(n):
    memo=[None]*(n+1)
    def cata_memo(k):
        if memo[k]==None:
            if k==0:
                memo[0]=1
            else:
                s=0
                for i in range(k):
                    s+=cata_memo(i)*cata_memo(k-1-i)
                memo[k]=s
        return memo[k]
    return cata_memo(n)
```

3. On saisit :

```
def cata4(n):
    tab=[1]+[0]*n
    for k in range(1,n+1):
        tab[k]=sum([tab[i]*tab[k-1-i] for i in range(k)])
    return tab[-1]
```

4. Comme pour la suite de Fibonacci, la version récursive sans mémoïsation est désastreuse, celle avec mémoïsation est meilleure mais n'égale pas la version ascendante.

Exercice 3

1. On saisit :

```
def binom1(n,k):
    """binom(n,k) descendant sans mémoisation"""
    if k==0:
        return 1
    elif k>n:
        return 0
    else:
        return binom1(n-1,k)+binom1(n-1,k-1)
```

2. On saisit :

```
def binom2(n,k):
    """binom(n,k) descendant avec mémoisation systématique"""
    memo={}
    def binom_memo(m,i):
        if (m,i) not in memo:
            if i==0:
                res=1
            elif i>m:
                res=0
            else:
                res=binom_memo(m-1,i)+binom_memo(m-1,i-1)
            memo[(m,i)]=res
        return memo[(m,i)]
    return binom_memo(n,k)
```

ou aussi, sans mémoïsation des cas de base :

```
def binom2(n,k):
    """binom(n,k) descendant avec memoisation
    sauf cas de base"""
    memo={}
    def binom_memo(m,i):
        if i==0:
            return 1
        elif i>m:
            return 0
        else:
            if (m,i) not in memo:
                memo[(m,i)]=binom_memo(m-1,i)+binom_memo(m-1,i-1)
            return memo[(m,i)]
    return binom_memo(n,k)
```

3. On saisit :

```
def binom3(n,k):
    tab=[[1]+[0]*n for i in range(n+1)]
    for i in range(1,n+1):
        for j in range(1,i+1):
            tab[i][j]=tab[i-1][j]+tab[i-1][j-1]
    return tab[n][k]
```

On récupère l'intégralité du triangle de Pascal en effectuant `return tab` à la dernière ligne.

4. On expérimente :

```
>>> res=binom3(10,1)
>>> for x in res:
        print(x)

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 3, 3, 1, 0, 0, 0, 0, 0, 0, 0]
[1, 4, 6, 4, 1, 0, 0, 0, 0, 0, 0]
[1, 5, 10, 10, 5, 1, 0, 0, 0, 0, 0]
...
```

Exercice 4

1. Pour la première ligne, on ne peut accéder à une case que depuis la case juste à gauche et pour la première colonne, on ne peut accéder à une case que depuis la case juste au-dessus. On en déduit :

$$\boxed{\forall j \in [\![1\,;\, \mathrm{C}-1]\!] \qquad t_{0,j} = 1 + t_{0,j-1} c_{0,j} \quad \text{et} \quad \forall i \in [\![1\,;\, \mathrm{L}-1]\!] \qquad t_{i,0} = 1 + t_{i-1,0} c_{i,0}}$$

2. Pour accéder à une case qui n'est pas sur la première ligne ou la première colonne, on peut arriver depuis la case juste au-dessus ou la case juste à gauche. On en déduit :

$$\boxed{\forall (i,j) \in [\![1\,;\, \mathrm{L}-1]\!] \times [\![1\,;\, \mathrm{C}-1]\!] \qquad t_{i,j} = 1 + \min(t_{i-1,j}, t_{i,j-1}) c_{i,j}}$$

3. On saisit :

```
def ppc_tab(config):
    L,C=len(config),len(config[0])
    tab=[[0]*C for k in range(L)]
    tab[0][0]=poids(0,0)
    for j in range(1,C):
        tab[0][j]=1+tab[0][j-1]*poids(0,j)
    for i in range(1,L):
        tab[i][0]=1+tab[i-1][0]*poids(i,0)
    for i in range(1,L):
        for j in range(1,C):
            tab[i][j]=1+min(tab[i-1][j],tab[i][j-1])*poids(i,j)
    return tab
```

4. On saisit :

```
def ppc(config):
    L,C=len(config),len(config[0])
    chemin=deepcopy(config)
    tab=ppc_tab(config)
    if tab[L-1][C-1]==float('inf'):
        return False
    i,j=L-1,C-1
    res=[[i,j]]
    while i>0 or j>0:
        if i>0 and tab[i][j]==1+tab[i-1][j]*poids(i,j):
            res.append([i-1,j])
            i-=1
        else:
            res.append([i,j-1])
            j-=1
        res.append([i,j])
    for i,j in res:
        chemin[i][j]=1
    return chemin,tab[L-1][C-1]-1
```

5. Le programme exécute l'algorithme A$\star$ et l'approche ascendante décrite ci-avant. On obtient par exemple :

```
A* - cout= 98
PPC dyn - cout= 98
```

avec les figures :

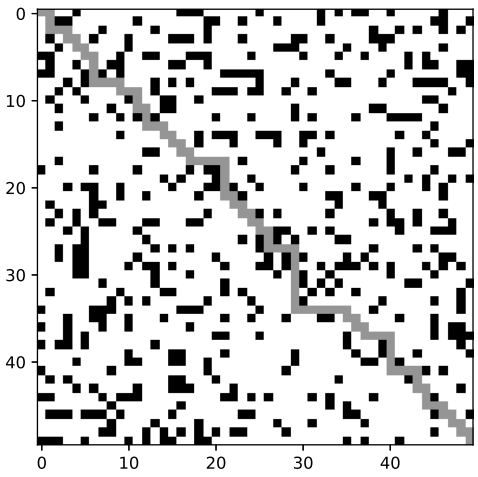


FIGURE 1 – A$\star$

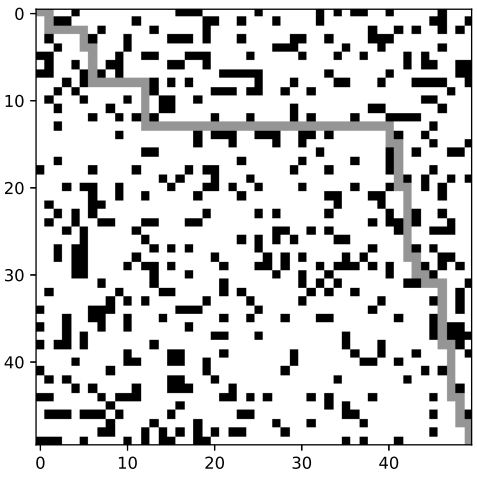


FIGURE 2 – Programmation dynamique

En réalisant plusieurs essais, on observe que la réalisation de A$\star$ peut échouer : la configuration étant générée aléatoirement, il n'est pas certain qu'il existe un chemin reliant $(0,0)$ à $(\mathrm{L}-1,\mathrm{C}-1)$,

auquel cas il n'existe évidemment pas de plus court chemin entre ces points. Par ailleurs, on peut également observer une situation où l'algorithme A$\star$ fonctionne mais pas la version ascendante : ceci arrive si tous les plus courts chemins de $(0, 0)$ à $(\mathrm{L}-1, \mathrm{C}-1)$ contiennent soit un déplacement vers le haut, soit un déplacement vers la gauche qui sont tous deux des déplacements interdits pour l'approche dynamique ascendante.

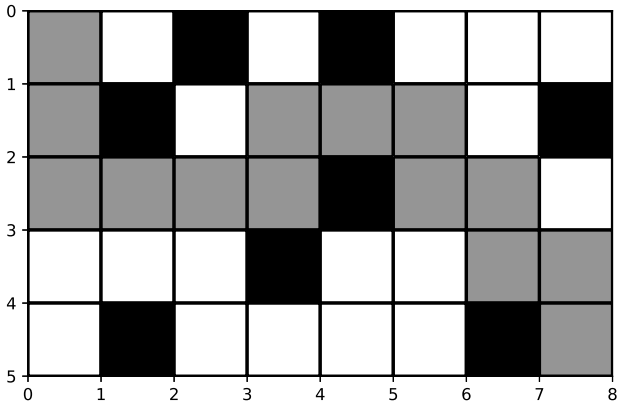


FIGURE 3 – Réussite de A$\star$ et échec de la programmation dynamique