Méthode « Diviser pour régner »

DEMERVILLE ERWAN

Rappels sur la récursivité

- La **récursivité** et l'**itération** exécutent plusieurs fois un ensemble d'instructions.

- On parle de fonction **récursive** lorsqu'une instruction dans le corps de la fonction **s'appelle elle-même**.
- On parle de fonction **itérative** lorsqu'une boucle s'exécute de manière répétée dans la fonction jusqu'à ce que la condition de contrôle devienne fausse.

Rappels sur la récursivité

	Récursivité	Itération
Signification	La fonction appelle elle-même.	Permet d'exécuter plusieurs fois l'ensemble des instructions.
Format	Dans une fonction récursive, seule la condition de terminaison est spécifiée.	L'itération comprend l'initialisation, la condition, l'exécution de l'instruction dans une boucle et la mise à jour (incrémenter et décrémenter) la variable de contrôle.
Terminaison	Une instruction conditionnelle est incluse dans le corps de la fonction pour forcer la fonction à retourner sans que l'appel de récurrence soit exécuté.	L'instruction d'itération est exécutée plusieurs fois jusqu'à ce qu'une certaine condition soit atteinte.
Condition	Si la fonction ne converge pas vers une condition appelée (cas de base), elle conduit à une récursion infinie.	Si la condition de contrôle dans l'instruction d'itération ne devient jamais fausse, elle conduit à une itération infinie.
Répétition infinie	Une récursion infinie peut bloquer le système.	Une boucle infinie utilise les cycles du processeur de manière répétée.
Application	La récursivité est toujours appliquée aux fonctions.	L'itération est appliquée aux instructions d'itération « ex : des boucles ».

Source : https://waytolearnx.com/2018/07/difference-entre-recursion-et-iteration.html

Rappels sur la récursivité

- On souhaite écrire une fonction calculant la factorielle d'un nombre entier n passé en entrée. Exemple : 4! = 1 x 2 x 3 x 4 = 24

✓ Méthode itérative :

```python
def factorielle_i(n: int) -> int:
    fact = 1
    # Boucle exécutant n - 1 itérations
    for i in range(2, n + 1):
        fact = fact * i
    return fact
```

Rappels sur la récursivité

- On souhaite écrire une fonction calculant la factorielle d'un nombre entier n passé en entrée. Exemple : 4! = 1 x 2 x 3 x 4 = 24

✓ Méthode récursive :

```python
def factorielle_r(n: int) -> int:
    # Cas de base
    if n == 1:
        return 1
    else:
        # Appel récursif
        return n * factorielle_r(n - 1)
```

Rappels sur la récursivité

- Exercice 1 : Voici une fonction. Que fait-elle ? Est-elle itérative ou récursive ?

```python
def somme(n):
    resultat = 0
    for i in range (n + 1):
        resultat = resultat + i*i
    return resultat
```

Rappels sur la récursivité

- Exercice 1 : Voici une fonction. Que fait-elle ? Est-elle itérative ou récursive ?

```python
def somme(n):
    resultat = 0
    for i in range (n + 1):
        resultat = resultat + i*i
    return resultat
```

- Cette fonction **calcule** la **somme des carrés** de **1** à **n**. Elle est **itérative**.
  Exemple : somme(5) = $1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$.

Rappels sur la récursivité

- Exercice 2 : Réécrire le programme suivant de manière récursive. Quel est le cas de base ?

```
def somme(n):
    resultat = 0
    for i in range (n + 1):
        resultat = resultat + i*i
    return resultat
```

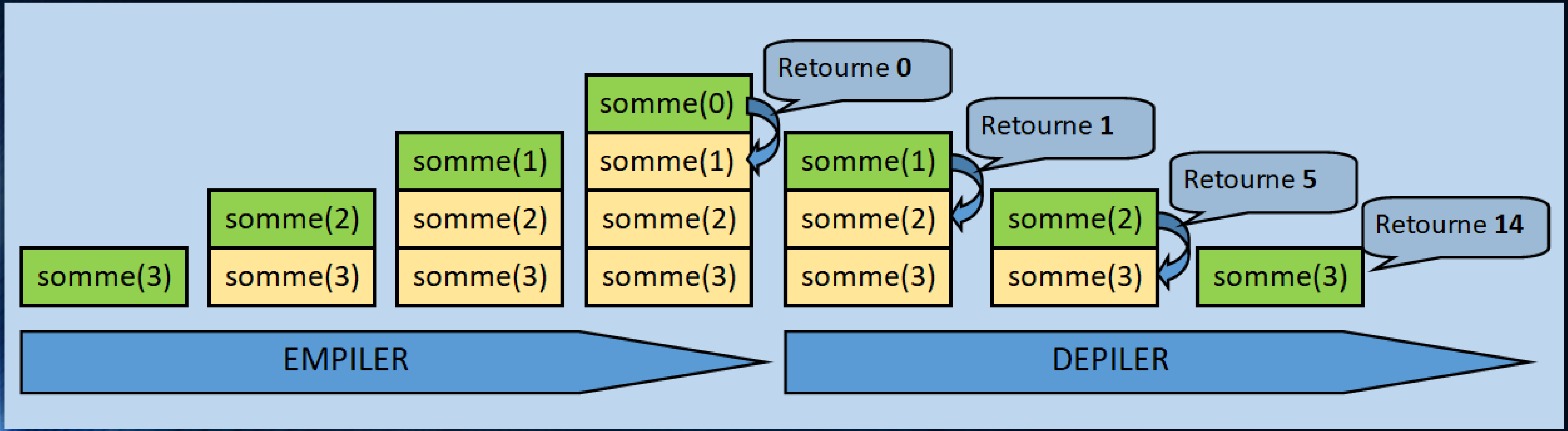
Rappels sur la récursivité

- Correction :

```python
def somme(n):
    if n == 0: # Condition d'arrêt
        return 0
    else:
        return somme(n - 1) + n*n
```

Rappels sur la récursivité

- Pour gérer des **fonctions récursives**, le système utilise une pile d'exécution pour stocker les informations sur les **fonctions en cours d'exécution** dans le programme.
- A chaque appel **récursif** d'une fonction, on empile la nouvelle fonction et on l'**exécute**, mettant en pause les fonctions en dessous dans la pile. A la fin de l'exécution, elle est dépilée et la fonction juste en dessous dans la pile poursuit son **exécution**.

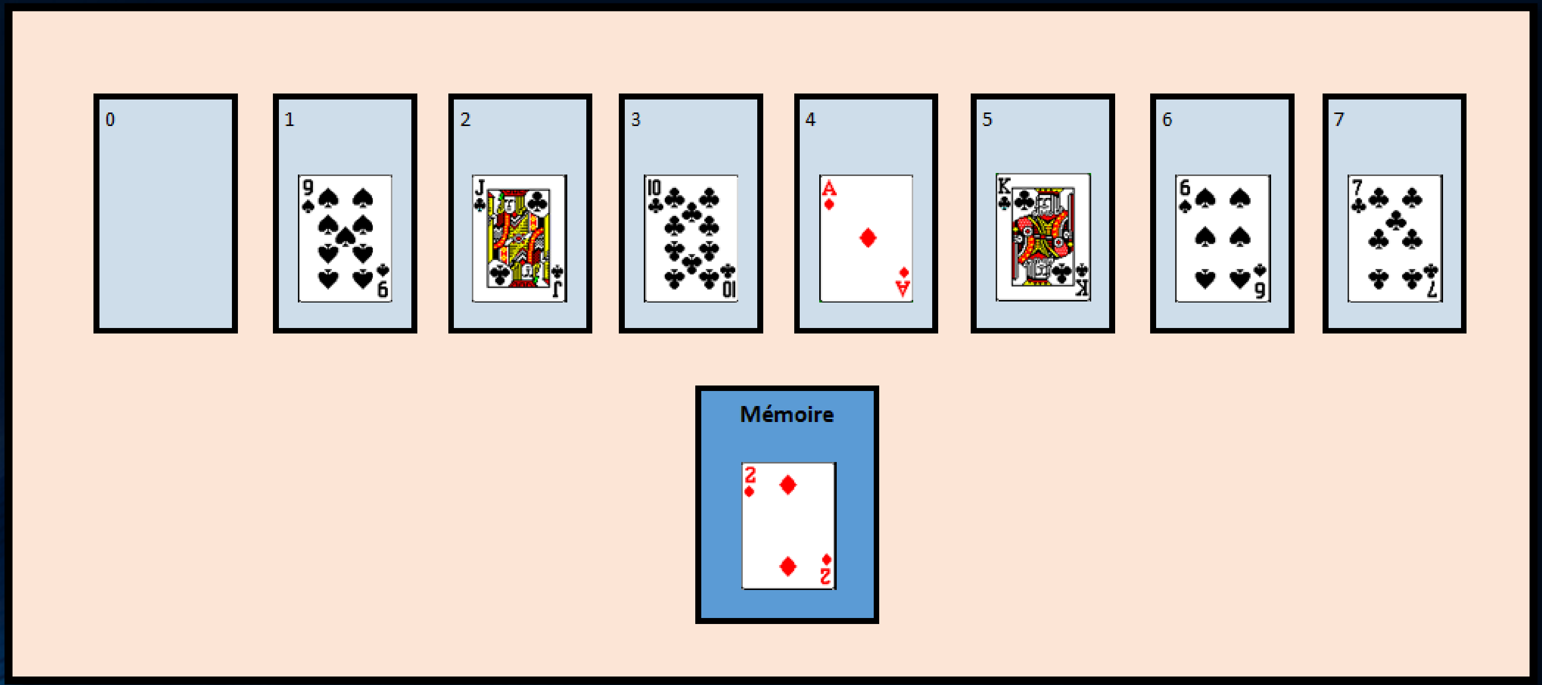


« Diviser pour régner »

- La méthode « diviser pour régner » consiste en 3 parties :
  - Diviser : Découper un problème initial en plusieurs sous-problèmes plus faciles à résoudre.
  - Régner : Résoudre les sous-problèmes, en général de manière récursive.
  - Combiner : Calculer une solution au problème initial en combinant les solutions des sous-problèmes.
- Cette méthode ramène la résolution d'un problème dépendant d'un entier n à la résolution d'un ou de plusieurs sous-problèmes indépendants dont la taille des entrées passe de n à n / 2 ou une fraction de n.
- Les algorithmes utilisant cette méthode s'écrivent de manière **récursive**, la taille du problème étant divisée à chaque appel récursif plutôt que seulement réduite d'une unité.

« Diviser pour régner »

- <u>Activité</u> : On souhaite rechercher la carte de valeur maximale parmi une liste de cartes. Comment feriez-vous naturellement ?
  - Prendre **8 cartes** au hasard. On considère les puissances des cartes dans cet ordre : As < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < V < D < R
  - La zone de travail est constituée de 8 emplacements pour les cartes, ainsi que d'un emplacement mémoire qui contient au début la **première carte**.

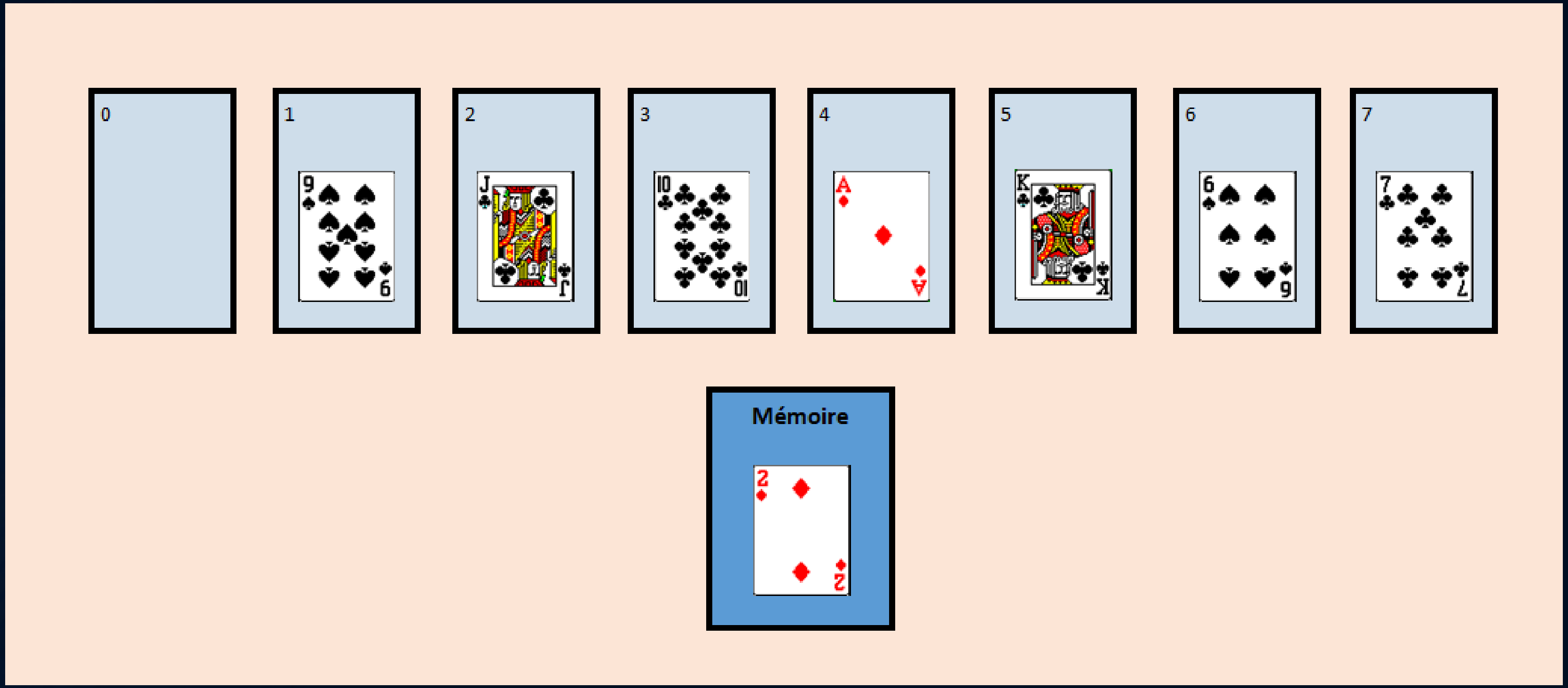


« Diviser pour régner »

- Activité : On souhaite rechercher la carte de valeur maximale parmi une liste de cartes. Comment feriez-vous naturellement ?
  - Prendre **8 cartes** au hasard. On considère les puissances des cartes dans cet ordre : As < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < V < D < R
  - La zone de travail est constituée de 8 emplacements pour les cartes, ainsi que d'un emplacement mémoire qui contient au début la **première carte**.
  - La lecture des cartes se fait de **gauche à droite**.
  - Vous ne pouvez effectuer qu'une action à la fois. Voici des exemples d'instructions possibles :
    - « Placer carte d'index i en mémoire » (retire l'ancienne carte de la mémoire)
    - « Comparer carte en mémoire avec carte d'index i »
    - « Carte mémoire < carte i, donc faire : »
    - « Retourner carte en mémoire »

« Diviser pour régner »

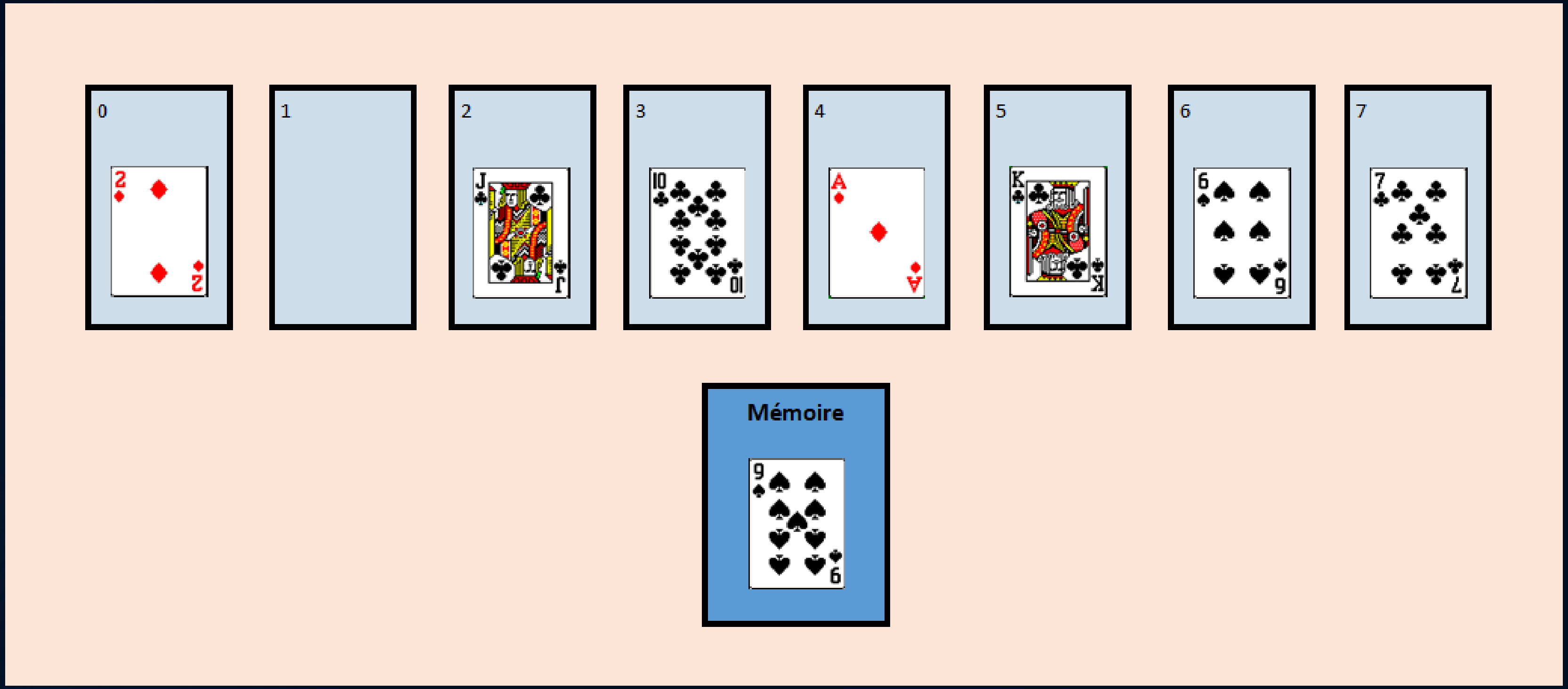
- Exemple d’application :



- Comparer carte en mémoire avec carte d’index 1.
- Carte en mémoire < carte 1, donc :
  - Mettre carte d’index 1 en mémoire.

« Diviser pour régner »

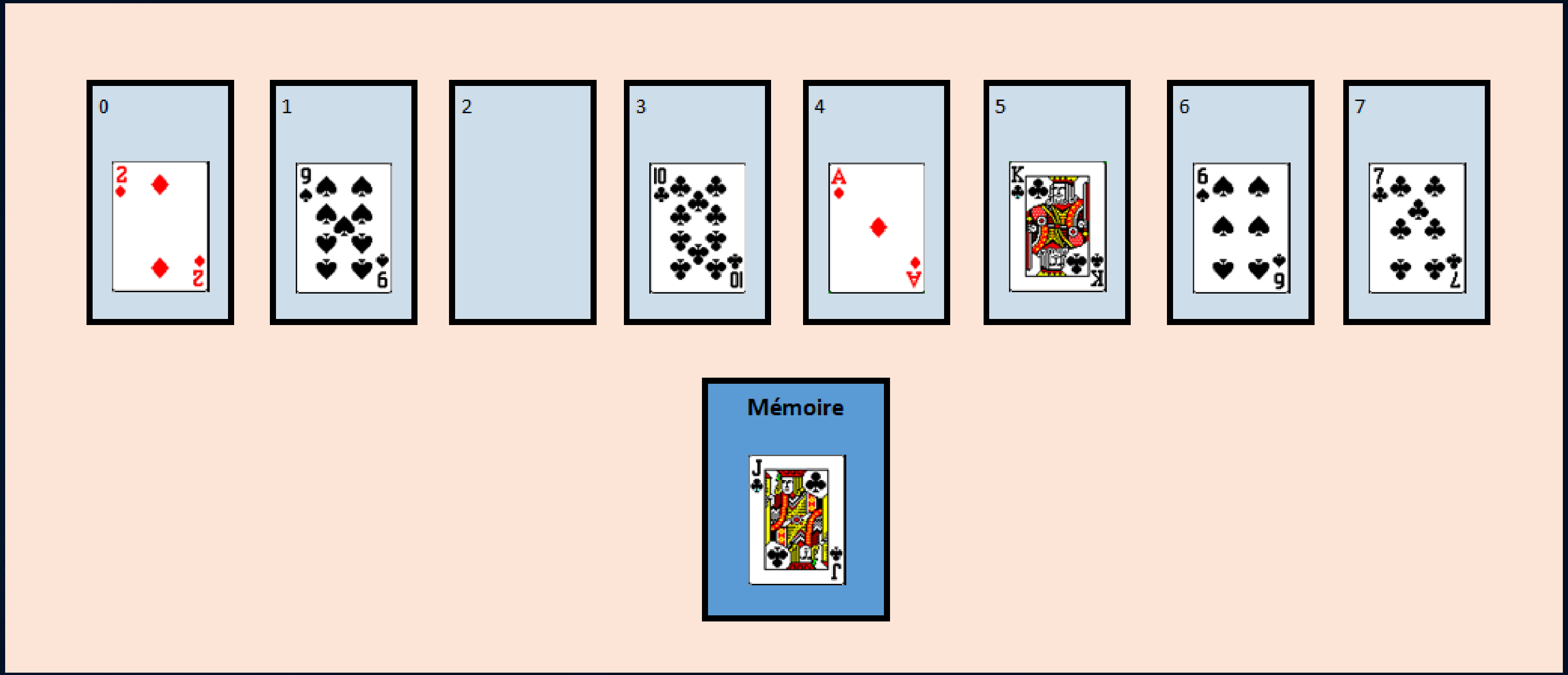
- Exemple d'application :



- Comparer carte en mémoire avec carte d'index 2.
- Carte en mémoire < carte 2, donc :
  - Mettre carte d'index 2 en mémoire.

« Diviser pour régner »

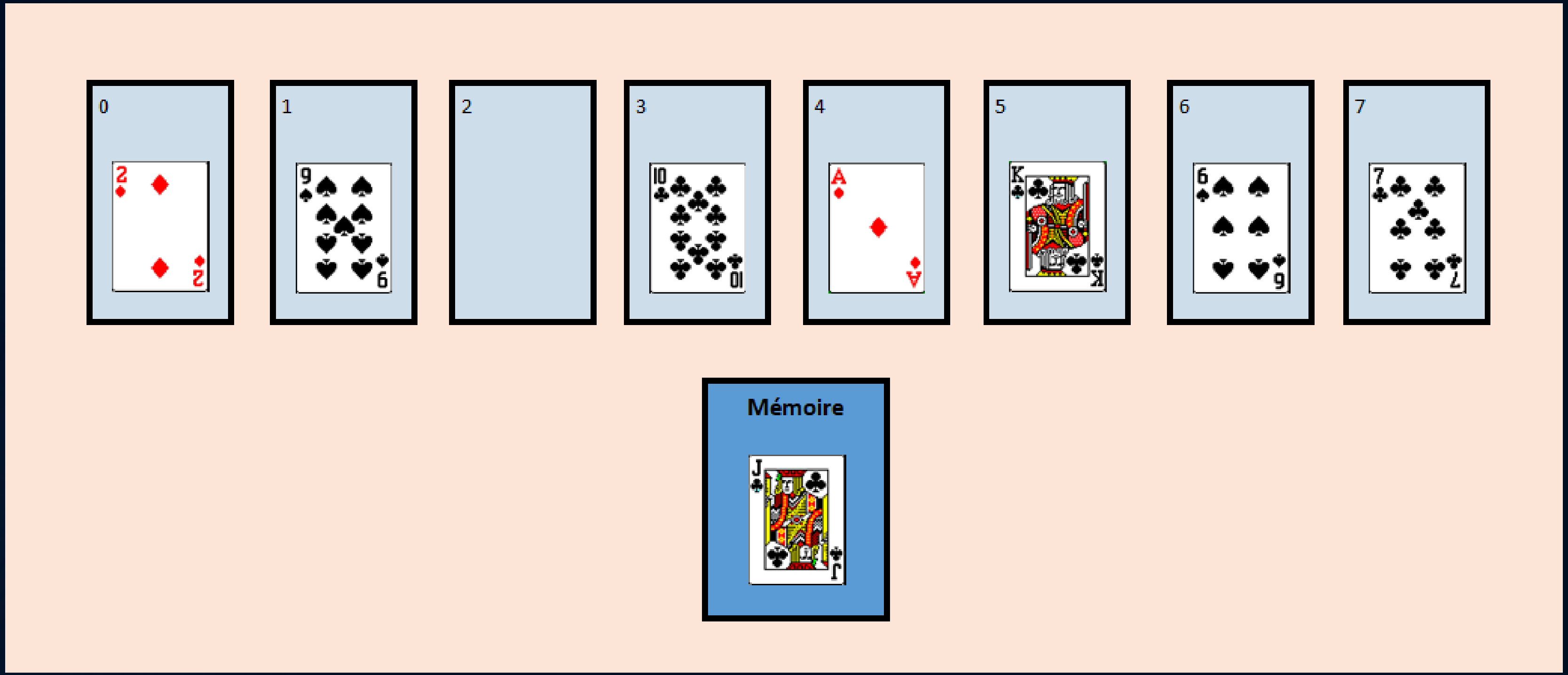
- <u>Exemple d'application</u> :



- Comparer carte en mémoire avec carte d'index 3.
- Carte en mémoire > carte 3, donc :
  - Ne rien faire.

« Diviser pour régner »

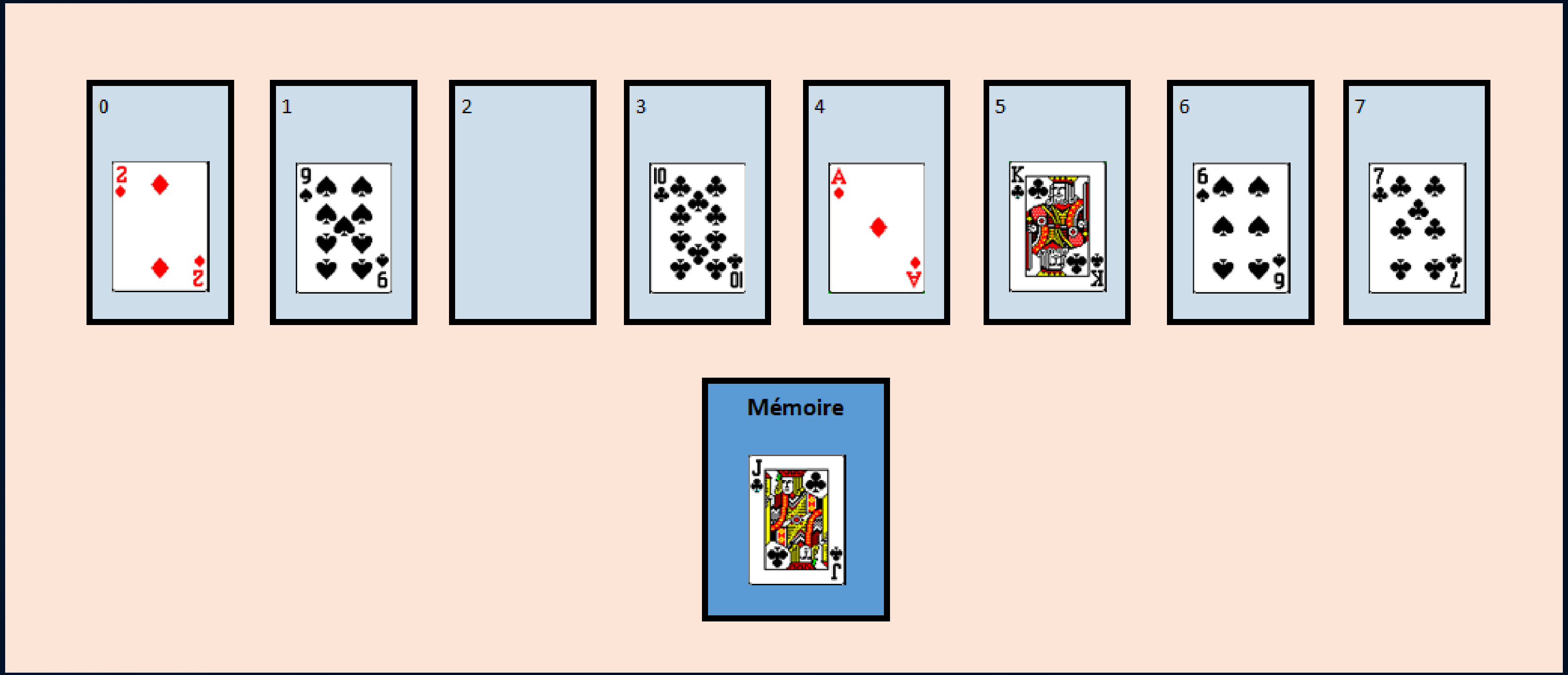
- Exemple d'application :



- Comparer carte en mémoire avec carte d'index 4.
- Carte en mémoire > carte 4, donc :
  - Ne rien faire.

« Diviser pour régner »

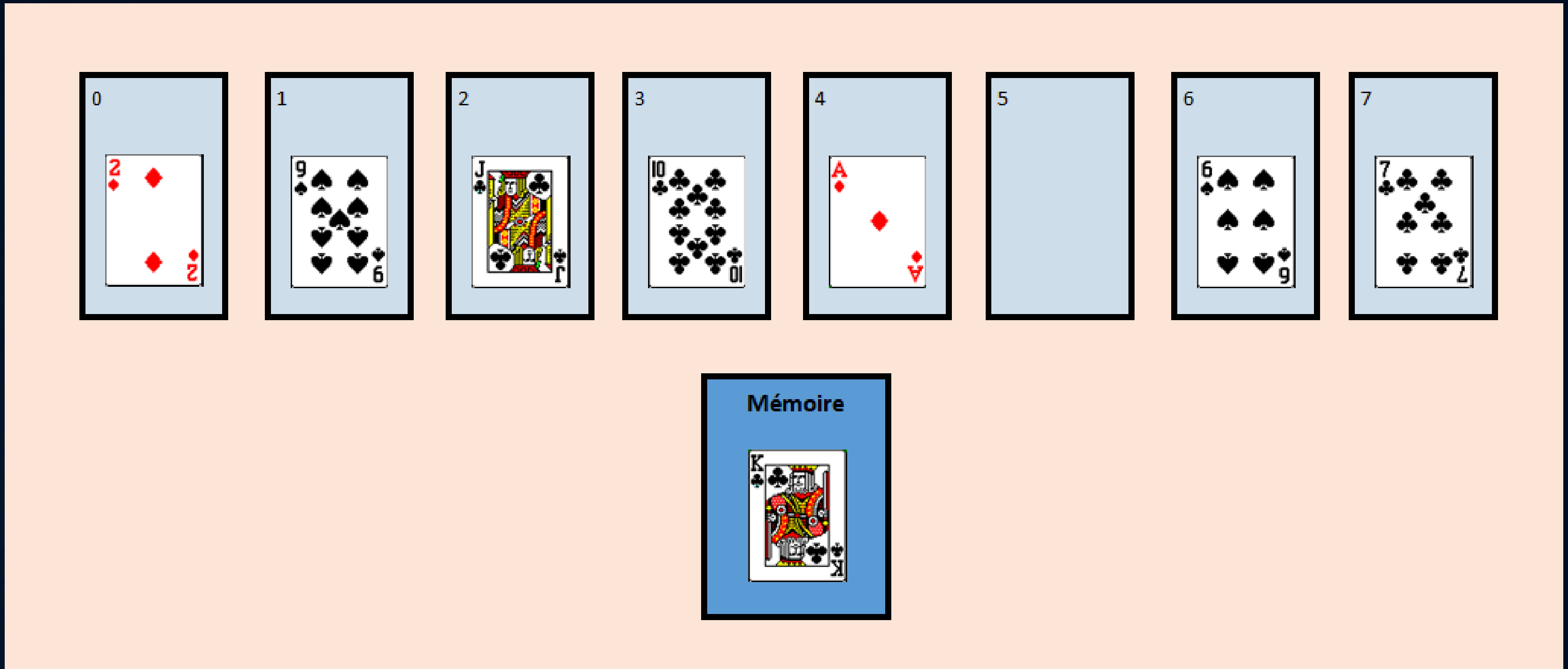
- Exemple d'application :



- Comparer carte en mémoire avec carte d'index 5.
- Carte en mémoire < carte 5, donc :
  - Mettre carte d'index 5 en mémoire.

« Diviser pour régner »

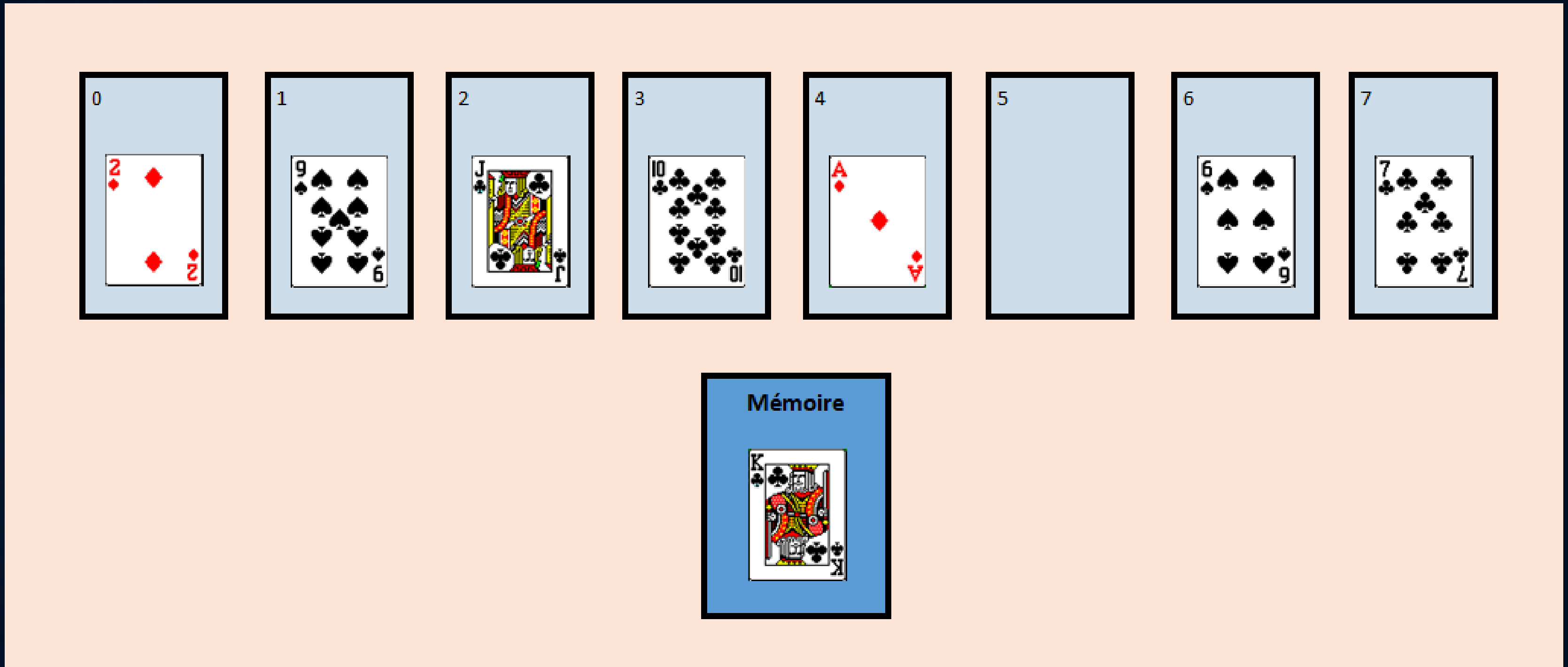
- Exemple d'application :



- Comparer carte en mémoire avec carte d'index 6.
- Carte en mémoire > carte 6, donc :
  - Ne rien faire.

« Diviser pour régner »

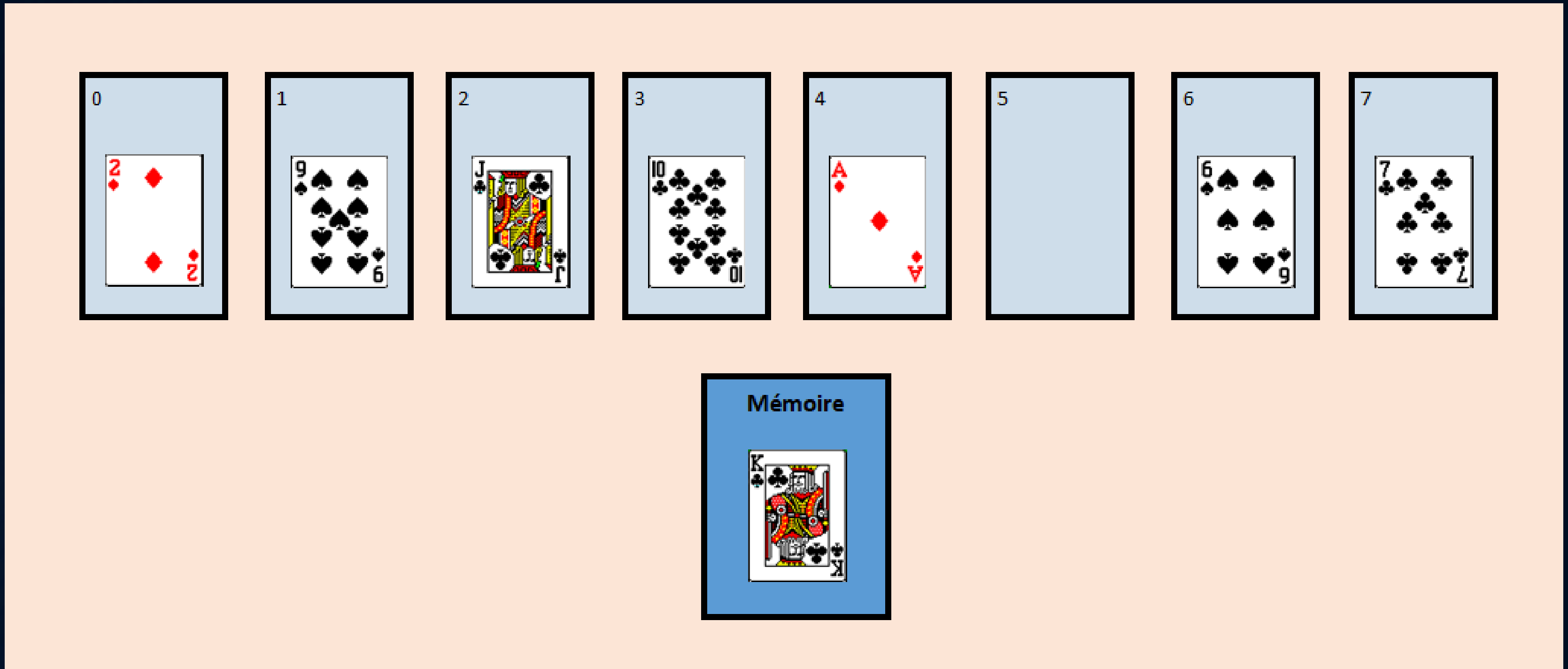
- <u>Exemple d'application</u> :



- Comparer carte en mémoire avec carte d'index 7.
- Carte en mémoire > carte 7, donc :
  - Ne rien faire.

« Diviser pour régner »

➢ <u>Exemple d'application</u> :



➢ Retourner carte en mémoire. Maximum = **Roi de trèfle** !

« Diviser pour régner »

- Exercice 3 : Ecrire une fonction Python maximum **itérative** qui retourne l'élément maximal d'une liste d'au moins 1 entier positif.

« Diviser pour régner »

- Exercice 3 : Ecrire une fonction Python maximum **itérative** qui retourne l'élément maximal d'une liste d'entiers positifs (avec n >= 1).

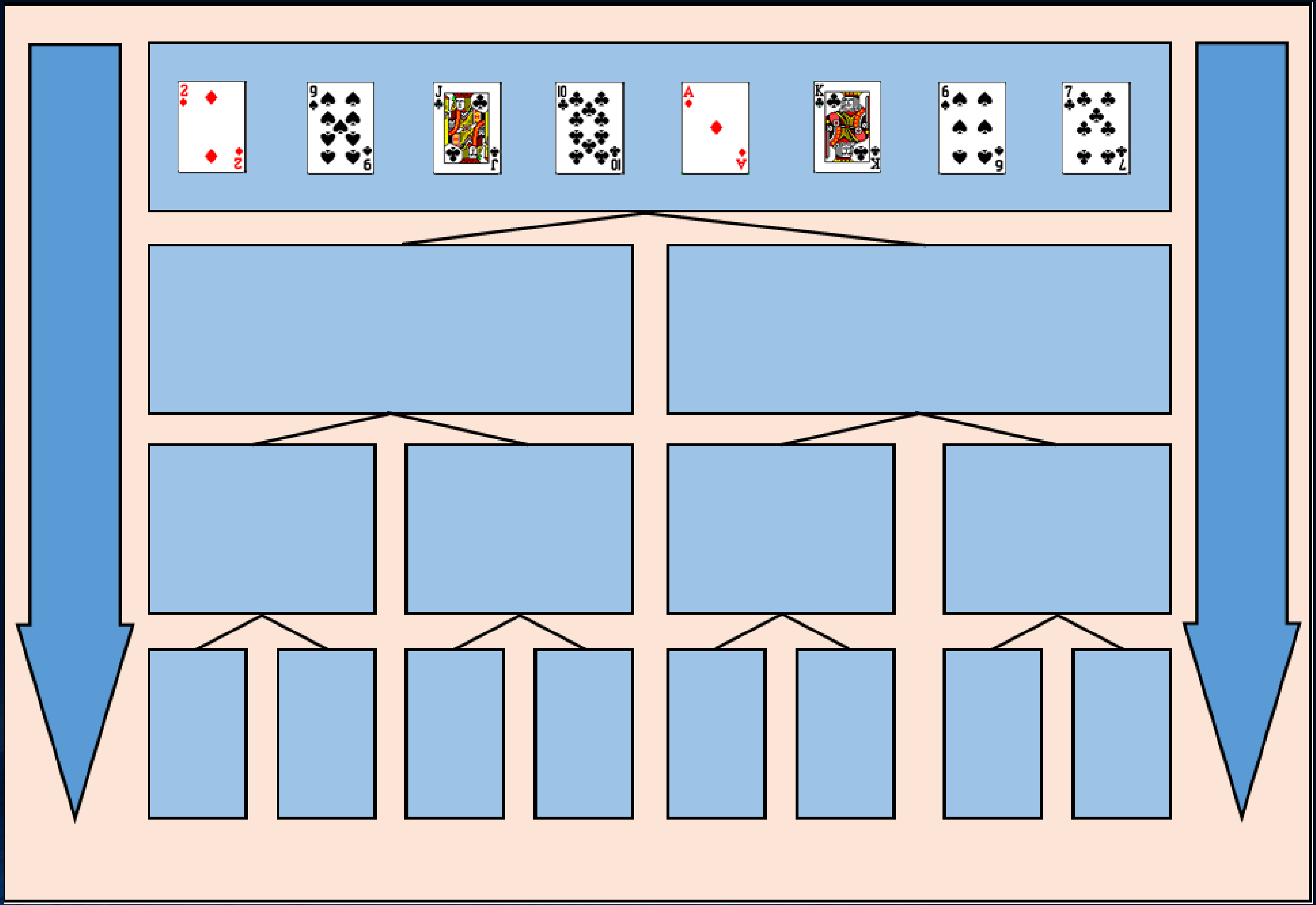
```python
def maximum(liste):
    max = liste[0]
    for i in range(1, len(liste)):
        if liste[i] > max:
            max = liste[i]
    return max

def maximum2(liste):
    max = 0
    for v in liste:
        if v > max:
            max = v
    return max
```

« Diviser pour régner »

- Activité : On souhaite rechercher la carte de valeur maximale parmi une liste de cartes, mais cette fois-ci à l'aide de la méthode « **diviser pour régner** ».

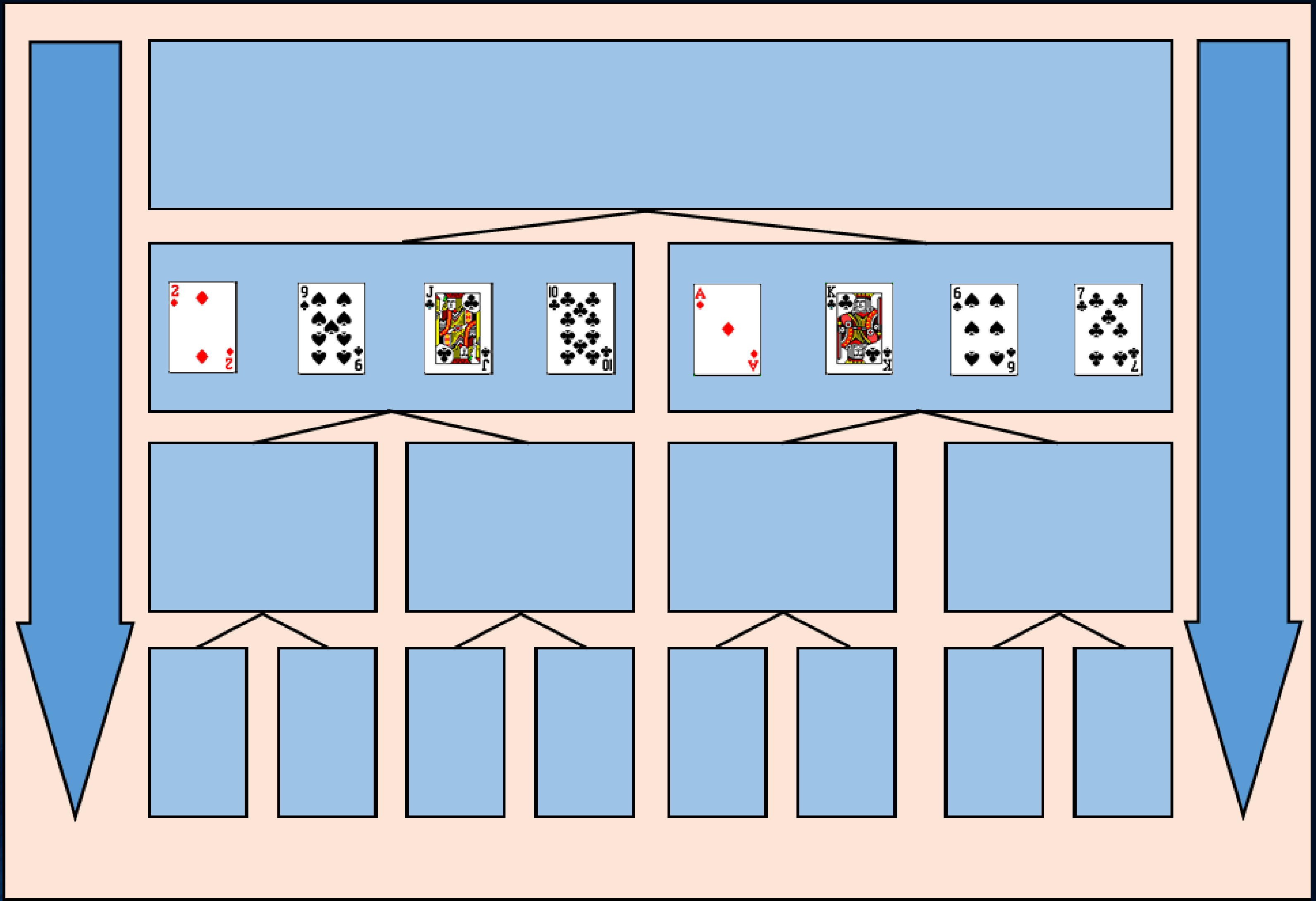
« Diviser pour régner »

- On divise la liste des cartes en **deux sous-listes** de **même taille**.



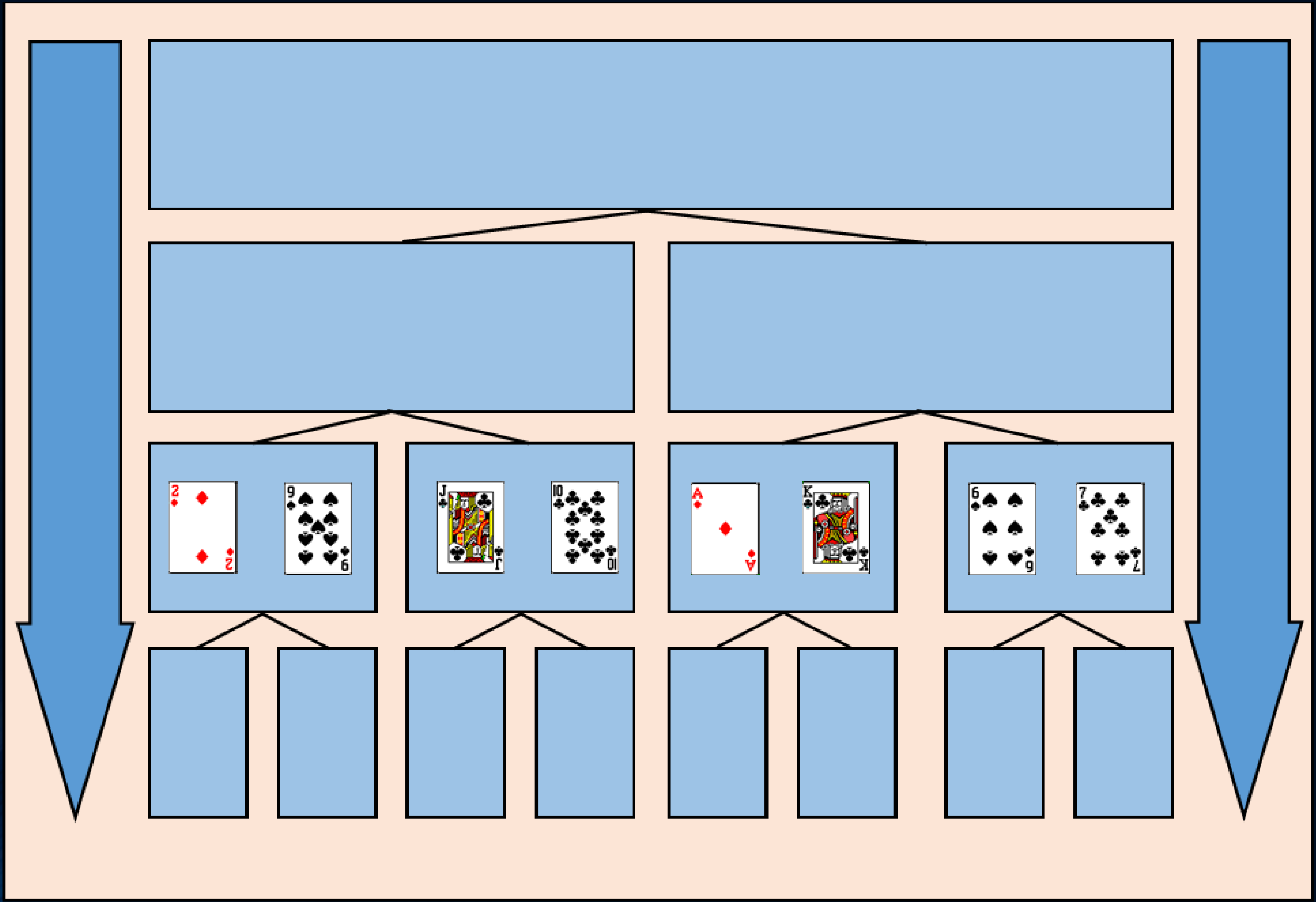
« Diviser pour régner »

- On divise de nouveau.



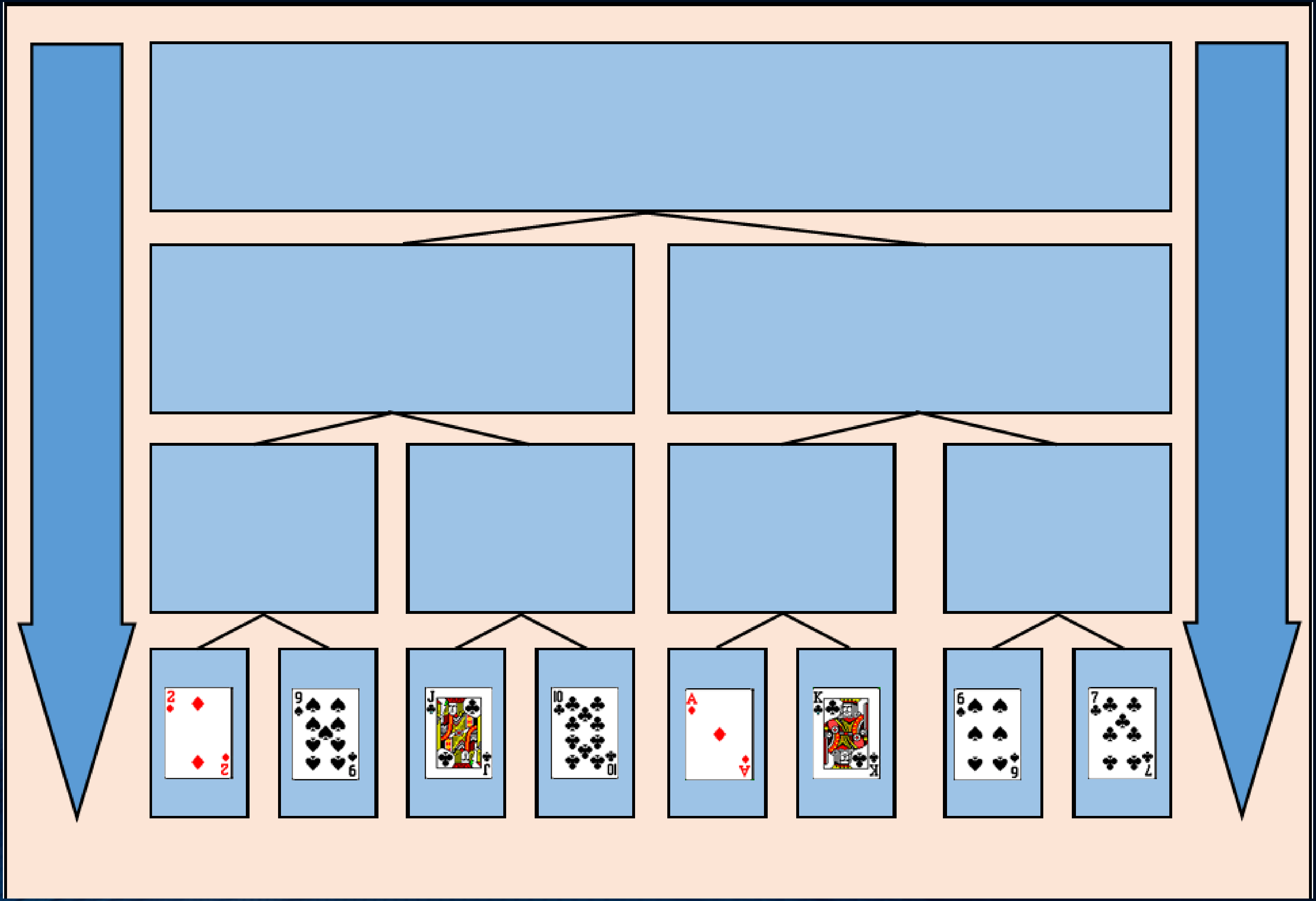
« Diviser pour régner »

- On divise de nouveau.



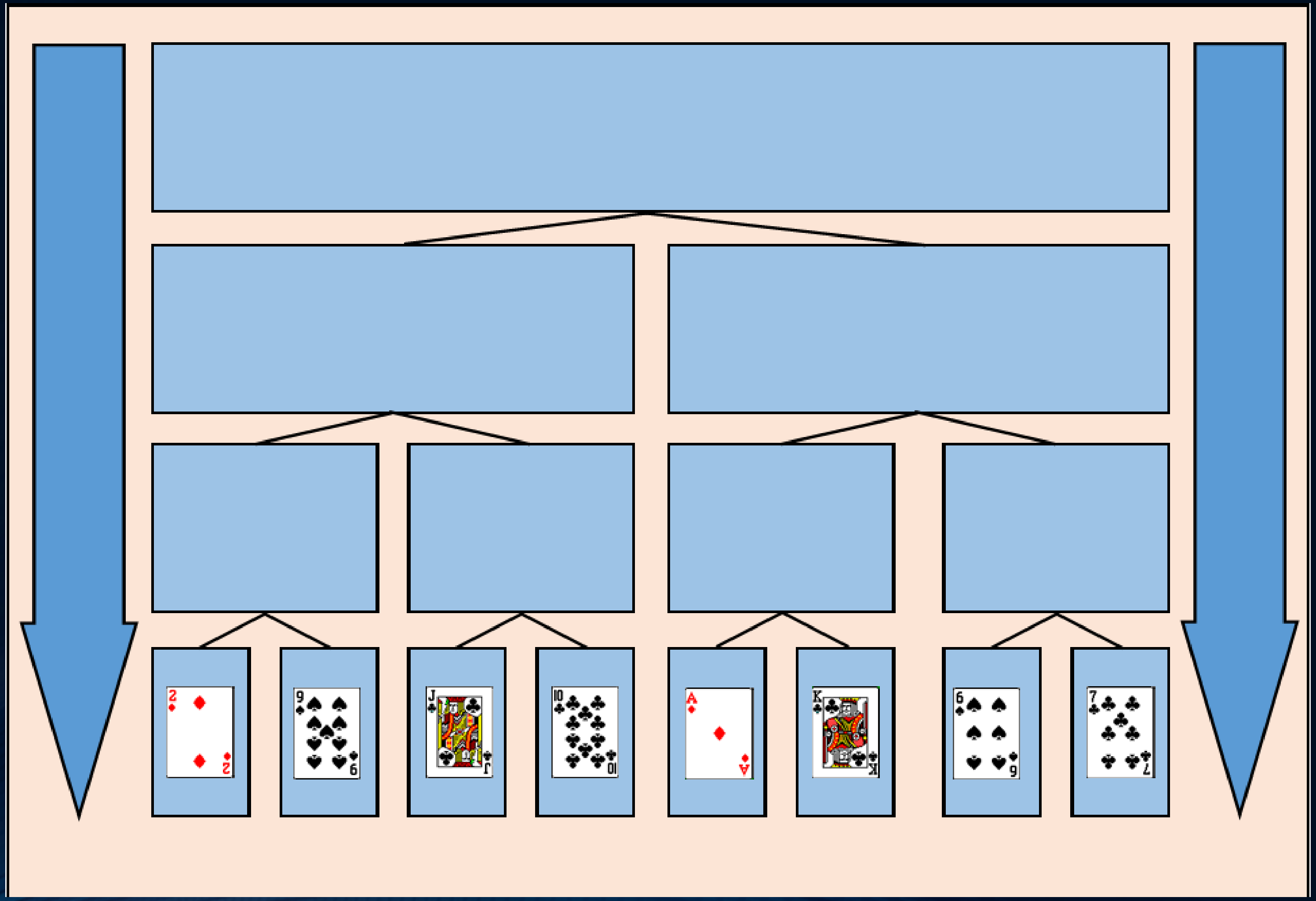
« Diviser pour régner »

- Maintenant, on remonte en gardant la plus haute carte pour chaque paire (on règne).



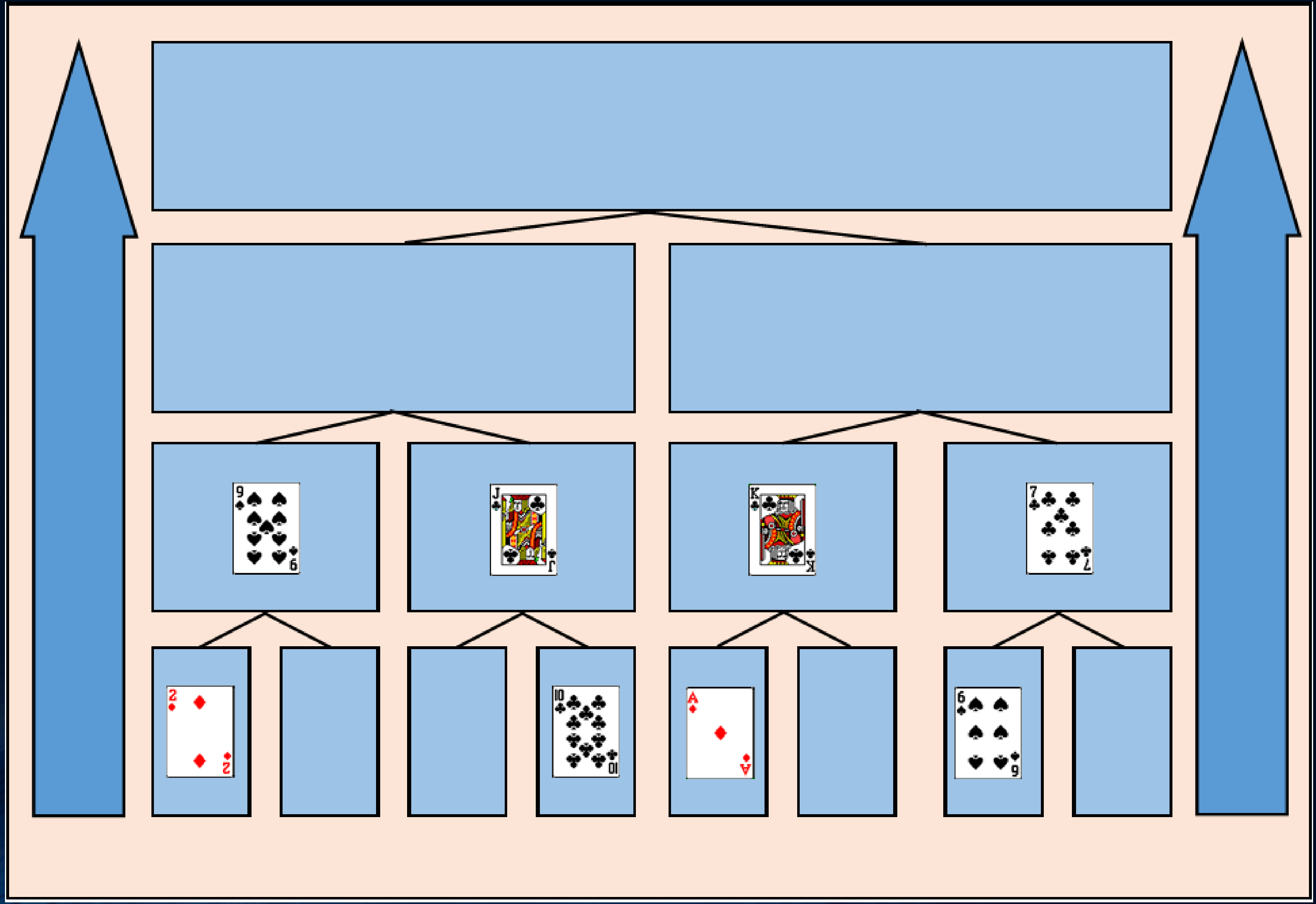
« Diviser pour régner »

- Chaque liste est constituée d’**une seule carte** : C’est donc la carte **la plus haute**. Max de [2] => 2, Max de [9] => 9, etc.



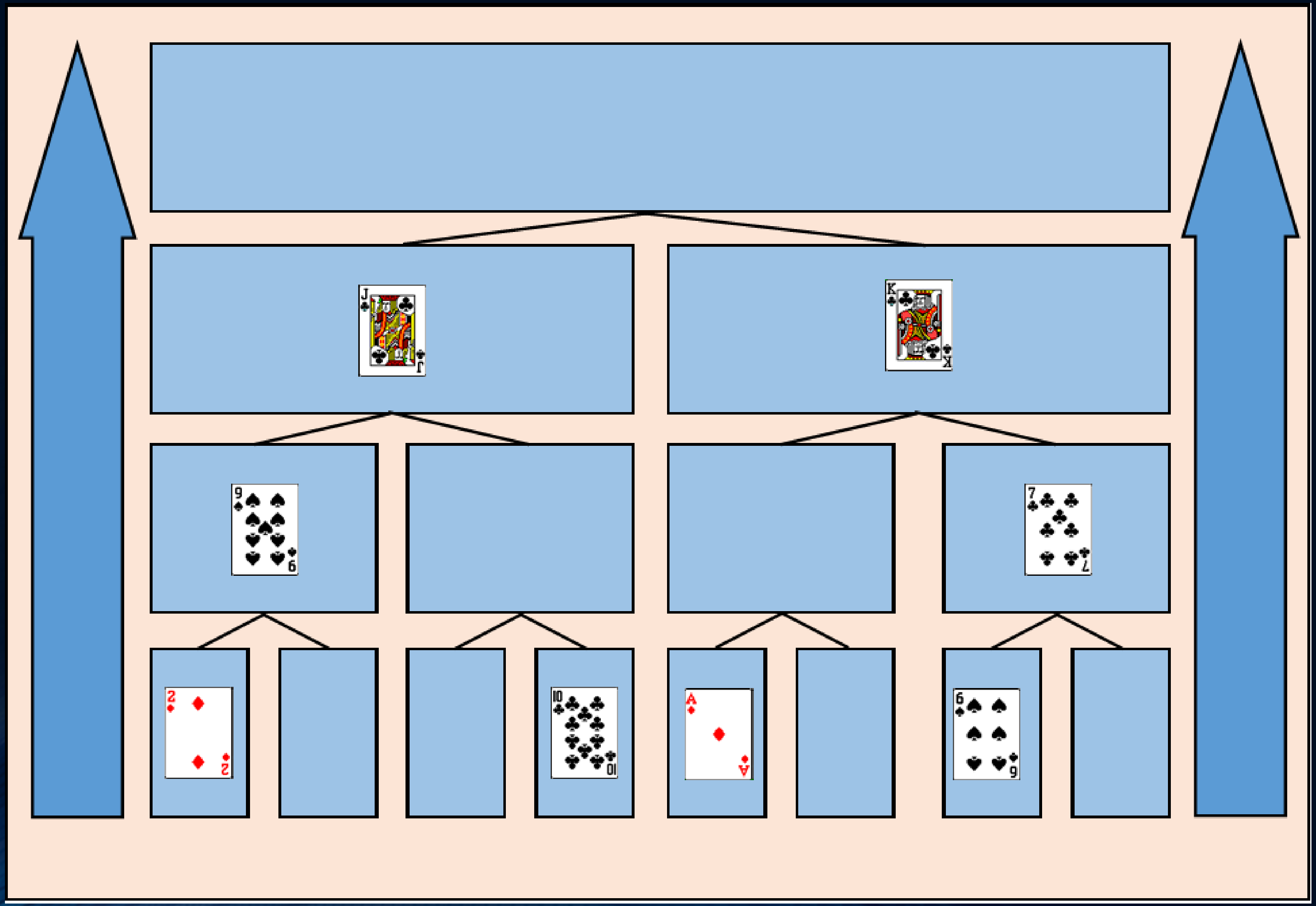
« Diviser pour régner »

- Max de [2, 9] => **9**. Max de [J, 10] => **J**. Max de [A, K] => **K**. Max de [6, 7] => **7**.



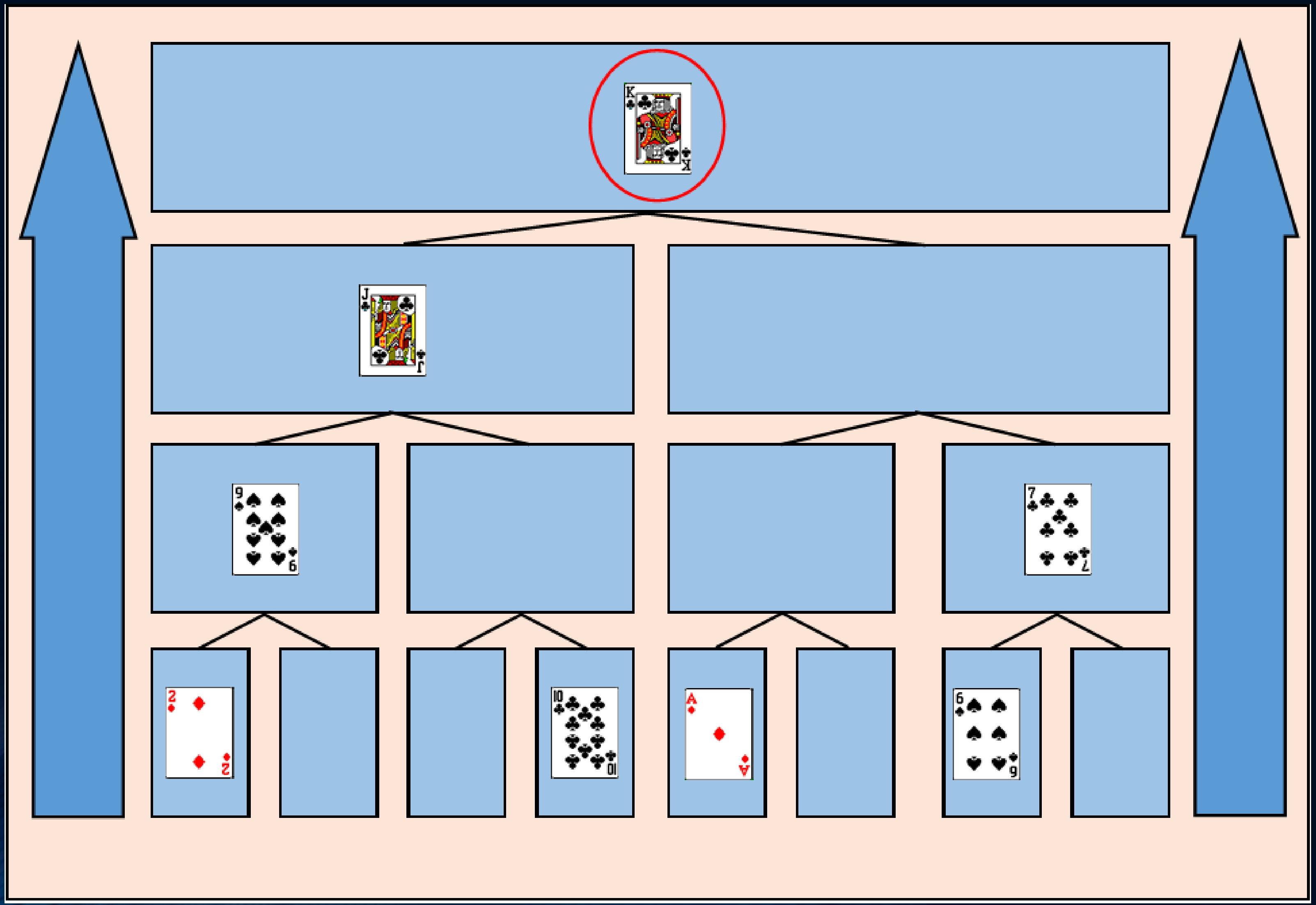
« Diviser pour régner »

- Max de [9, J] => J. Max de [K, 7] => K.



« Diviser pour régner »

- Max de [J, K] => K. Le **roi de trèfle** est la carte **la plus haute** !



« Diviser pour régner »

- Activité : On souhaite rechercher la carte de valeur maximale parmi une liste de cartes, mais cette fois-ci à l'aide de la méthode « **diviser pour régner** ».
- Choisissez 8 cartes au hasard.
- La lecture des cartes s'effectue cette fois-ci de **haut en bas** et de **bas en haut.**
- Adaptez la méthode « diviser pour régner » (Diviser, régner, combiner). Notez chaque action que vous effectuez pour parvenir à trouver la **carte la plus haute**.

« Diviser pour régner »

- Activité : On souhaite rechercher la carte de valeur maximale parmi une liste de cartes, mais cette fois-ci à l'aide de la méthode « **diviser pour régner** ».

- Choisissez 8 cartes au hasard.

- La lecture des cartes s'effectue cette fois-ci de **haut en bas** et de **bas en haut.**

  - En descendant : A chaque descente d'un cran, **séparer** la liste des cartes en **deux sous-listes de même taille**. Diviser les nouvelles **sous-listes** en **deux sous-listes**… Et ainsi de suite jusqu'à ce que **chaque liste ne contienne qu'une seule carte**.

  - En remontant : A chaque montée d'un cran, ne garder que la carte de **valeur la plus haute** (**régner**) parmi **chaque couple** de cartes, et **recombiner** ainsi les nouvelles listes.

- Notez chaque action que vous effectuez pour parvenir à trouver la carte de valeur maximale.

« Diviser pour régner »

- Exercice 4 : On souhaite écrire une fonction Python maximum_r **récursive** et utilisant la méthode « **diviser pour régner** » qui retourne l'élément maximal d'une liste constituée d'au moins 1 entier positif.

- Ecrire d'abord une fonction plus_grand qui renvoie la valeur la plus grande entre deux entiers.

« Diviser pour régner »

- Exercice 4 : On souhaite écrire une fonction Python maximum_r **récursive** et utilisant la méthode « **diviser pour régner** » qui retourne l'élément maximal d'une liste constituée d'au moins 1 entier positif.

- Ecrire d'abord une fonction plus_grand qui renvoie la valeur la plus grande entre deux entiers.

```python
def plus_grand(elt1, elt2):
    if elt1 > elt2:
        return elt1
    else:
        return elt2
```

« Diviser pour régner »

- Exercice 4 : Ecrire une fonction Python maximum_r **récursive** et utilisant la méthode « **diviser pour régner** » qui retourne l'élément maximal d'une liste constituée d'au moins 1 entier positif.

Voici la liste des instructions de l'algorithme pour vous aider :

- Si la taille de la liste vaut 1, on retourne le seul élément de la liste. (cas de base)
- Sinon :
  - On stocke la moitié gauche de la liste dans une variable gauche.
  - On stocke la moitié droite de la liste dans une variable droite.
  - On renvoie la valeur la plus grande entre le max de gauche et le max de droite.

« Diviser pour régner »

➤ <u>Correction</u> :

```python
def plus_grand(elt1, elt2):
    if elt1 > elt2:
        return elt1
    else:
        return elt2

def maximum_r(liste):
    if len(liste) == 1:
        return liste[0]
    else:
        # On définit gauche et droite par compréhension :
        gauche = [liste[i] for i in range(0, len(liste)//2)]
        droite = [liste[i] for i in range(len(liste)//2, len(liste))]
        return plus_grand(maximum_r(gauche), maximum_r(droite))
```

- Ici, le **cas de base** se produit lorsque la liste n'est constituée que d'un seul élément, auquel cas on retourne l'élément en question.

Vers de nouvelles méthodes de tri

- On propose de découvrir une **nouvelle méthode de tri** plus efficace que les tris vus en première (le tri sélection et le tri insertion).
- Rappels : On rappelle l'algorithme du tri par insertion ainsi que sa complexité :

```python
def tri_insertion(tab):
    taille = len(tab)
    for j in range(1, taille):
        cle = tab[j]
        i = j - 1
        while i >= 0 and tab[i] > cle:
            tab[i + 1] = tab[i]
            i = i - 1
        tab[i + 1] = cle
```

- Nombre de comparaisons :
  - Meilleur des cas : n – 1 comparaisons (Complexité linéaire)
  - Pire des cas : $\frac{n(n-1)}{2} = 1 + 2 + \cdots + (n-1)$ comp. (Complexité quadratique)

Vers de nouvelles méthodes de tri

- On propose de découvrir une **nouvelle méthode de tri** plus efficace que les tris vus en première (le tri sélection et le tri insertion).

- Rappels : On rappelle l'algorithme du tri par sélection ainsi que sa complexité :

```python
def tri_selection(tab):
    # Pour chaque valeur du tableau :
    for i in range(len(tab)):

        # Trouver le minimum parmi les valeurs suivantes
        min = i
        for j in range(i+1, len(tab)):
            if tab[min] > tab[j]:
                min = j

        # Echanger la valeur à l'indice i avec la valeur min
        tmp = tab[i]
        tab[i] = tab[min]
        tab[min] = tmp
```

- Nombre de comparaisons :
  - Dans **tous les cas** : $\frac{n(n-1)}{2} = 1 + 2 + \cdots + (n-1)$ comp. (Complexité quadratique)

Vers de nouvelles méthodes de tri

- Exercice 5 :

On souhaite écrire une fonction **récursive** fusion qui retourne la **fusion** de **deux listes triées**.

  - Quels sont les **deux** cas de base ?
    Rappel : Les cas de base sont nécessaires pour que l’algorithme se termine.

Vers de nouvelles méthodes de tri

- Exercice 7 :

On souhaite écrire une fonction **récursive** fusion qui retourne la **fusion** de **deux listes triées**.

- Quels sont les **deux** cas de base ?
  Rappel : Les cas de base sont nécessaires pour que l'algorithme se termine.

Cas 1 : La liste 1 est vide, auquel cas on retourne la liste 2.

Cas 2 : La liste 2 est vide, auquel cas on retourne la liste 1.

- Quels sont les deux autres cas à identifier ?

Vers de nouvelles méthodes de tri

- Exercice 7 :

On souhaite écrire une fonction **récursive** fusion qui retourne la **fusion** de **deux listes triées**.

- Quels sont les **deux** cas de base ?
  Rappel : Les cas de base sont nécessaires pour que l'algorithme se termine.

Cas 1 : La liste 1 est vide, auquel cas on retourne la liste 2.

Cas 2 : La liste 2 est vide, auquel cas on retourne la liste 1.

- Quels sont les **deux autres cas** à identifier ?

Cas 3 : Le premier élément de la liste 1 est inférieur au premier élément de la liste 2. On retourne la concaténation du premier élément de la liste 1 et de la fusion du reste de la liste 1 avec la liste 2.

Cas 4 : Dernier cas, le premier élément de la liste 1 est supérieur au premier élément de la liste 2. On retourne la concaténation du premier élément de la liste 2 et de la fusion de la liste 1 avec le reste de la liste 2.

Vers de nouvelles méthodes de tri

➢ Exercice 7 :

Ecrire maintenant la fonction Python **récursive** fusion qui retourne la **fusion** de **deux listes triées**. On rappelle les 4 cas :

Cas 1 : La liste 1 est vide, auquel cas on retourne la liste 2.

Cas 2 : La liste 2 est vide, auquel cas on retourne la liste 1.

Cas 3 : Le premier élément de la liste 1 est inférieur au premier élément de la liste 2. On retourne la concaténation du premier élément de la liste 1 et de la fusion du reste de la liste 1 avec la liste 2.

Cas 4 : Dernier cas, le premier élément de la liste 1 est supérieur au premier élément de la liste 2. On retourne la concaténation du premier élément de la liste 2 et de la fusion de la liste 1 avec le reste de la liste 2.

Vers de nouvelles méthodes de tri

➢ Correction exercice 7 :

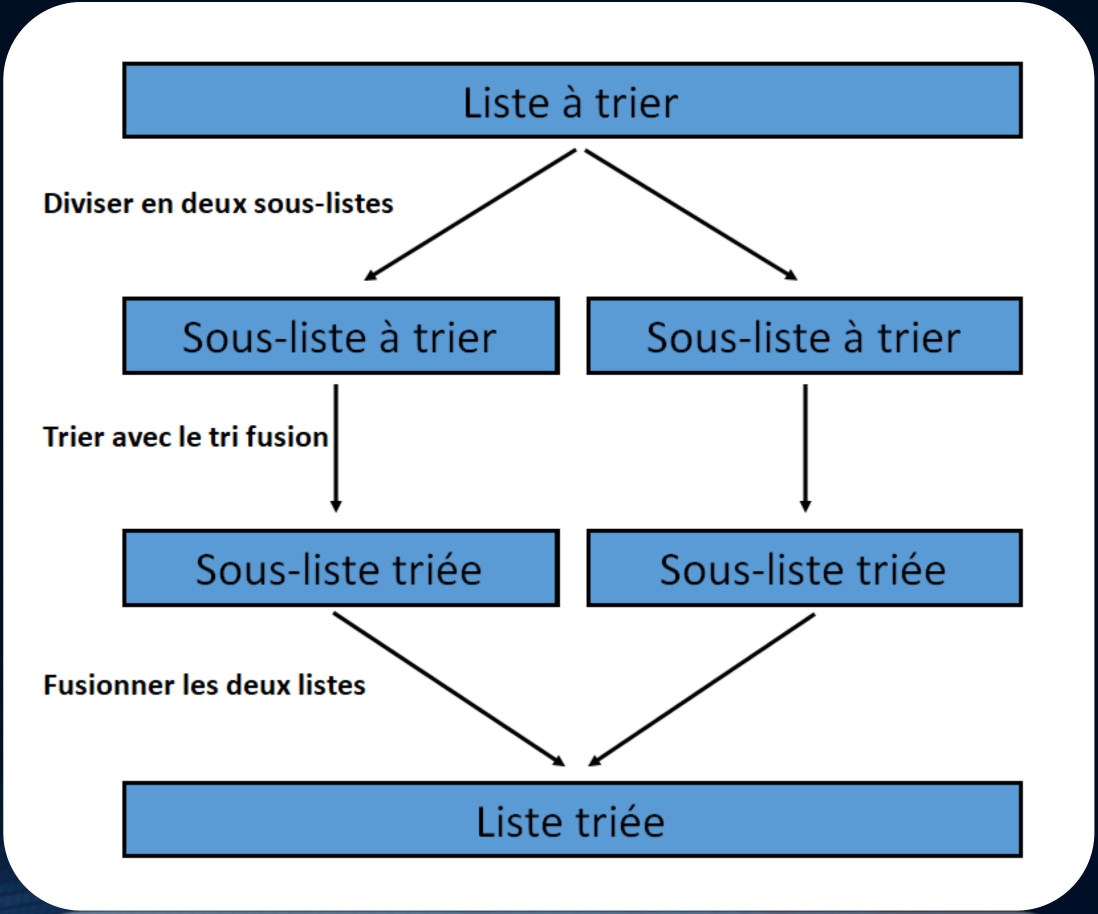
Ecrire maintenant la fonction Python **récursive** fusion qui retourne la **fusion** de **deux listes triées**.

```python
def fusion(L1, L2):
    """ Fonction retournant une liste triée, fusion de deux listes triées.
    :param L1: (list d'int) Liste d'entiers triée
    :param L2: (list d'int) Liste d'entiers triée
    :return: (list d'int) Fusion des deux listes triées """
    global NB_COMP

    if L1 == []:
        return L2
    elif L2 == []:
        return L1
    elif L1[0] < L2[0]:
        return [L1[0]] + fusion([L1[i] for i in range(1, len(L1))], L2)
    else:
        return [L2[0]] + fusion(L1, [L2[i] for i in range(1, len(L2))])
```

Vers de nouvelles méthodes de tri

➤ <u>Exercice 7 suite</u> :

On souhaite à présent écrire la fonction Python tri_fusion, faisant appel à la fonction fusion, et qui effectue le **tri de deux listes**.



Vers de nouvelles méthodes de tri

- Exercice 7 suite :

Ecrire maintenant la fonction Python tri_fusion, faisant appel à la fonction fusion, et qui effectue le **tri de deux listes**.

```python
def tri_fusion(Liste):
    """ Effectue le tri fusion de la liste passée en entrée.
    :param Liste: (list d'int) Liste d'entiers
    :return: (list d'int) Une liste d'entiers triée """

    n = len(Liste)
    if n <= 1:
        return Liste
    else:
        m = n // 2
        return fusion(tri_fusion([Liste[i] for i in range(0, m)]),
                      tri_fusion([Liste[i] for i in range(m, n)]))
```

Vers de nouvelles méthodes de tri

- On souhaite analyser la complexité du tri fusion, en comparaison avec le tri insertion et le tri sélection.

- A noter : Le tri fusion **récursif** est plus facile à écrire que la version **itérative**, mais a un coût **spatial** (= en **mémoire**) plus élevé (linéaire) car il nécessite la recopie de la liste pour créer les deux sous-listes.

➢ Quel est le nombre de comparaisons effectué par la fonction fusion, en fonction des tailles N1 et N2 des deux listes ?

  ➢ Dans le meilleur cas ?

  ➢ Dans le pire des cas ?

  ➢ Si N1 = N2 ?

  ➢ Si N1 != N2 ?

Vers de nouvelles méthodes de tri

- On souhaite analyser la complexité du tri fusion, en comparaison avec le tri insertion et le tri sélection.
- Quel est le nombre de comparaisons effectué par la fonction fusion, en fonction des tailles N1 et N2 des deux listes ?

➢ Dans le meilleur cas ?

  ➢ Que N1 soit égal ou non à N2, on a toujours N1 comparaisons.

➢ Dans le pire des cas ?

  ➢ Si N1 = N2 : (N2 + N1) – 1 comparaisons.

  ➢ Si N1 > N2 : $N2 * 2$ comparaisons.

  ➢ Si N2 > N1 : $2N1 - 1$ comparaisons.

Vers de nouvelles méthodes de tri

- On souhaite analyser la complexité du tri fusion, en comparaison avec le tri insertion et le tri sélection.

➢ Essayez de compléter le tableau suivant en indiquant le nombre de comparaisons effectuées par la fonction tri_fusion selon les cas :

Cas numéro	Liste	Nombre de comparaisons
1	T = [1, 2, 3, 4]	
2	T = [1,3,2,4]	
3	T = [8,7,6,5,4,3,2,1]	
4	T = [1,5,3,7,2,6,4,8]	

➢ Quels sont, parmi ces 4 cas, les pires cas ? Les meilleurs cas ?

Vers de nouvelles méthodes de tri

- On souhaite analyser la complexité du tri fusion, en comparaison avec le tri insertion et le tri sélection.

➢ Essayez de compléter le tableau suivant en indiquant le nombre de comparaisons effectuées par la fonction tri_fusion selon les cas :

Cas numéro	Liste	Nombre de comparaisons
1	T = [1, 2, 3, 4]	**4**
2	T = [1,3,2,4]	**5**
3	T = [8,7,6,5,4,3,2,1]	**12**
4	T = [1,5,3,7,2,6,4,8]	**17**

➢ Quels sont, parmi ces 4 cas, les pires cas ? Les meilleurs cas ?

- Les meilleurs cas sont lorsque la liste est déjà triée, que ça soit de manière croissante ou décroissante (cas 1 et 3). Les pires cas sont lorsque les nombres sont répartis comme dans les cas 2 et 4.

Vers de nouvelles méthodes de tri

- On souhaite analyser la complexité du tri fusion, en comparaison avec le tri insertion et le tri sélection.

- Le nombre de comparaisons du tri fusion dans le pire des cas est de :

$comp(n) = comp(\frac{n}{2}) + \mathrm{comp}(\frac{n}{2}) + \mathrm{n} - 1$ si n est pair

$comp(n) = comp(\frac{n}{2}) + \mathrm{comp}(\frac{n}{2}) + \mathrm{n} - 2$ si n est impair

$\mathrm{n} - 1$ et $\mathrm{n} - 2$ sont d'ordre linéaire, on peut donc approximer la complexité :

$C(n) = C(\frac{n}{2}) + \mathrm{C}(\frac{n}{2}) + \mathrm{O(n)}$

La complexité temporelle du tri fusion est en O(n log(n)) dans tous les cas.

On parle de complexité quasi-linéaire, donc un peu moins efficace qu'un coût linéaire.

En comparaison avec les tris sélection et insertion qui sont quadratiques (dans tous les cas pour le tri sélection, dans le pire cas pour le tri insertion), on peut en conclure que le tri fusion est, en **comparaison**, **plus efficace dans l'ensemble**.

Le tri insertion est toutefois plus efficace que le tri fusion lorsque la liste est déjà triée.

FIN