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 :

```
1  def factorielle_i(n: int) -> int:
2      fact = 1
3      # Boucle exécutant n - 1 itérations
4      for i in range(2, n + 1):
5          fact = fact * i
6      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 :

```
1  def factorielle_r(n: int) -> int:
2      # Cas de base
3      if n == 1:
4          return 1
5      else:
6          # Appel récursif
7          return n * factorielle_r(n - 1)
```

Rappels sur la récursivité

- Exercice 1 : Voici un programme. Que fait-il ? Est-il itératif ou récursif ?

```
1 n = int(input("Saisir un nombre entier: "))
2
3 def somme(n):
4     resultat = 0
5     for i in range (n):
6         resultat = resultat + i*i
7     return resultat
8
9 print("le résultat est:", somme(n+1))
```

Rappels sur la récursivité

- Exercice 1 : Voici un programme. Que fait-il ? Est-il itératif ou récursif ?

```
1  n = int(input("Saisir un nombre entier: "))
2
3  def somme(n):
4      resultat = 0
5      for i in range (n):
6          resultat = resultat + i*i
7      return resultat
8
9  print("le résultat est:", somme(n+1))
```

- Ce programme **demande** à l'utilisateur de **saisir** un **entier**, puis **affiche** la **somme des carrés** de **1** à **n**. Il est **itératif**.
  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.

```
1  n = int(input("Saisir un nombre entier: "))
2
3  def somme(n):
4      resultat = 0
5      for i in range (n):
6          resultat = resultat + i*i
7      return resultat
8
9  print("le résultat est:", somme(n+1))
```

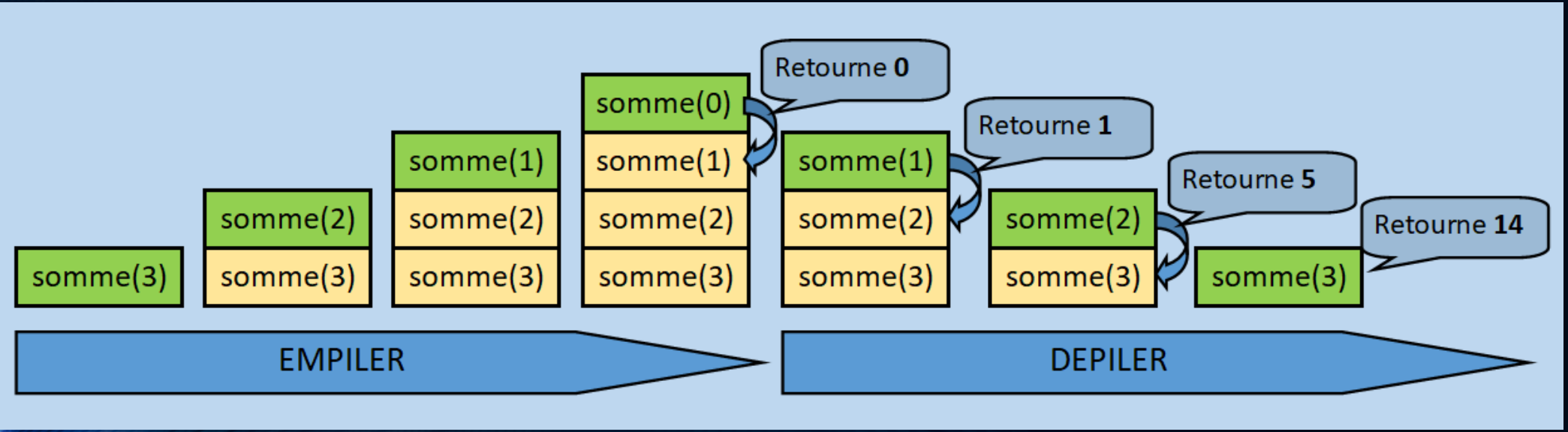
Rappels sur la récursivité

- Correction :

```python
n = int(input("Saisir un nombre entier: "))

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

print("le résultat est:", somme(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**.



Rappels sur la récursivité

- Exercice 3 :

  La suite de Fibonacci est telle que : $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n > 1$.
  Chaque terme de la suite de Fibonacci est la somme des deux termes qui le précèdent.

  - Exemple : $\mathbf{F_4} = F_3 + F_2 = (F_2 + F_1) + (F_1 + F_0) = ((F_1 + F_0) + F_1) + (F_1 + F_0) = ((1 + 0) + 1) + (1 + 0) = \mathbf{3}$

  Ecrire une fonction fibonacci récursive donnant le n-ième terme de la suite de Fibonacci.

Rappels sur la récursivité

- Correction :

Ecrire une fonction fibonacci récursive donnant le n-ième terme de la suite de Fibonacci :

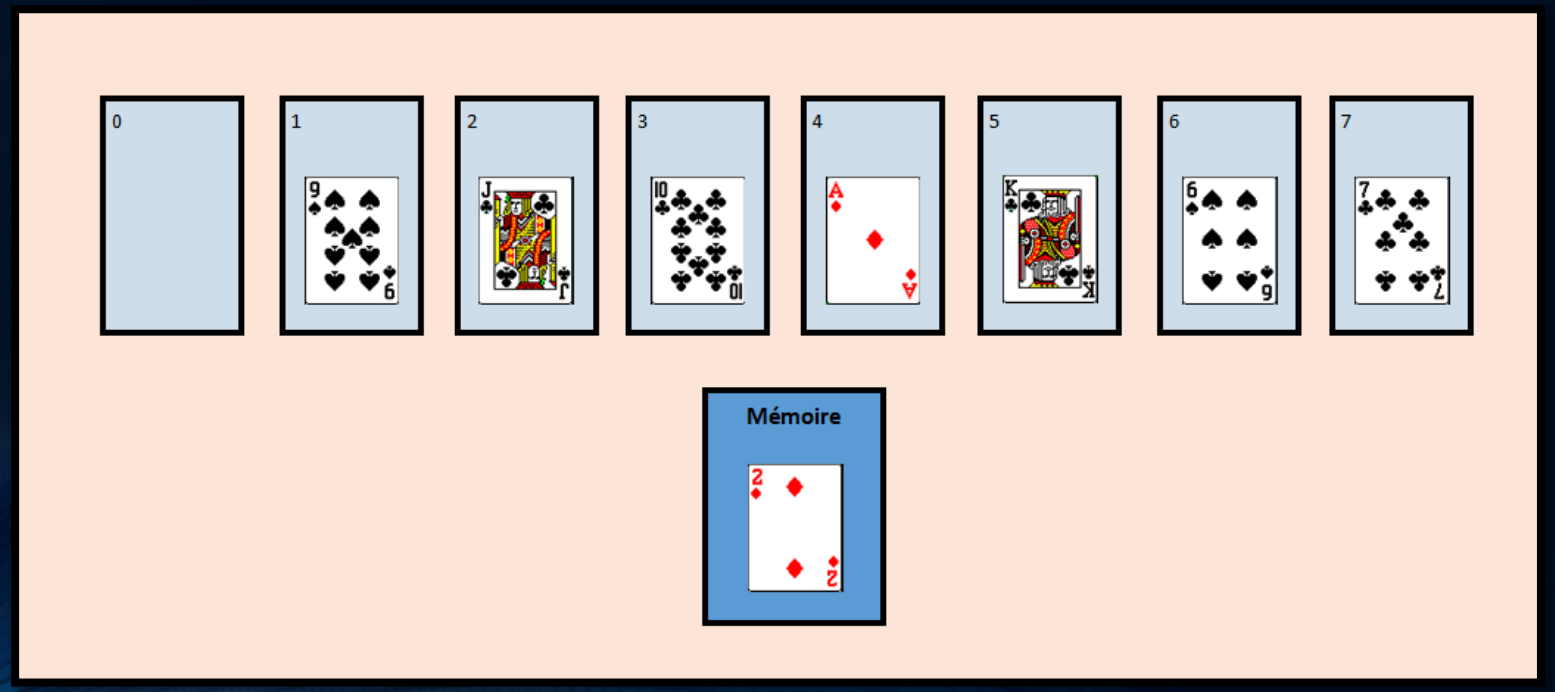
```python
def fibonacci(n: int) -> int:
    """ Retourne le n-ième terme de la suite de Fibonacci.
    :param n: (int) Numéro du terme souhaité de la suite
    :return: (int) n-ième terme de la suite de Fibonacci
    Condition d'utilisation : n >= 0
    Conditions d'arrêt : n = 0 ou n = 1 """

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

« 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 »

- 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**.

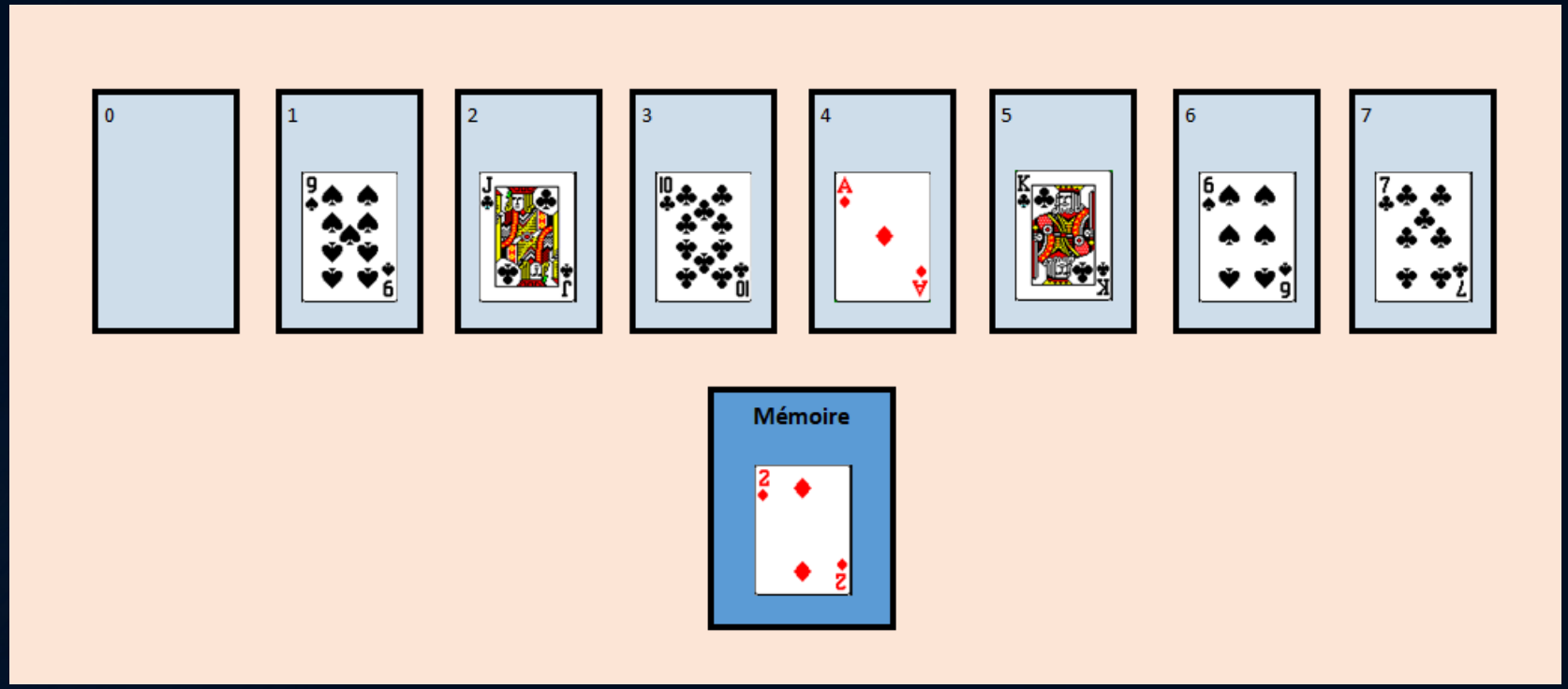


« Diviser pour régner »

- 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 »

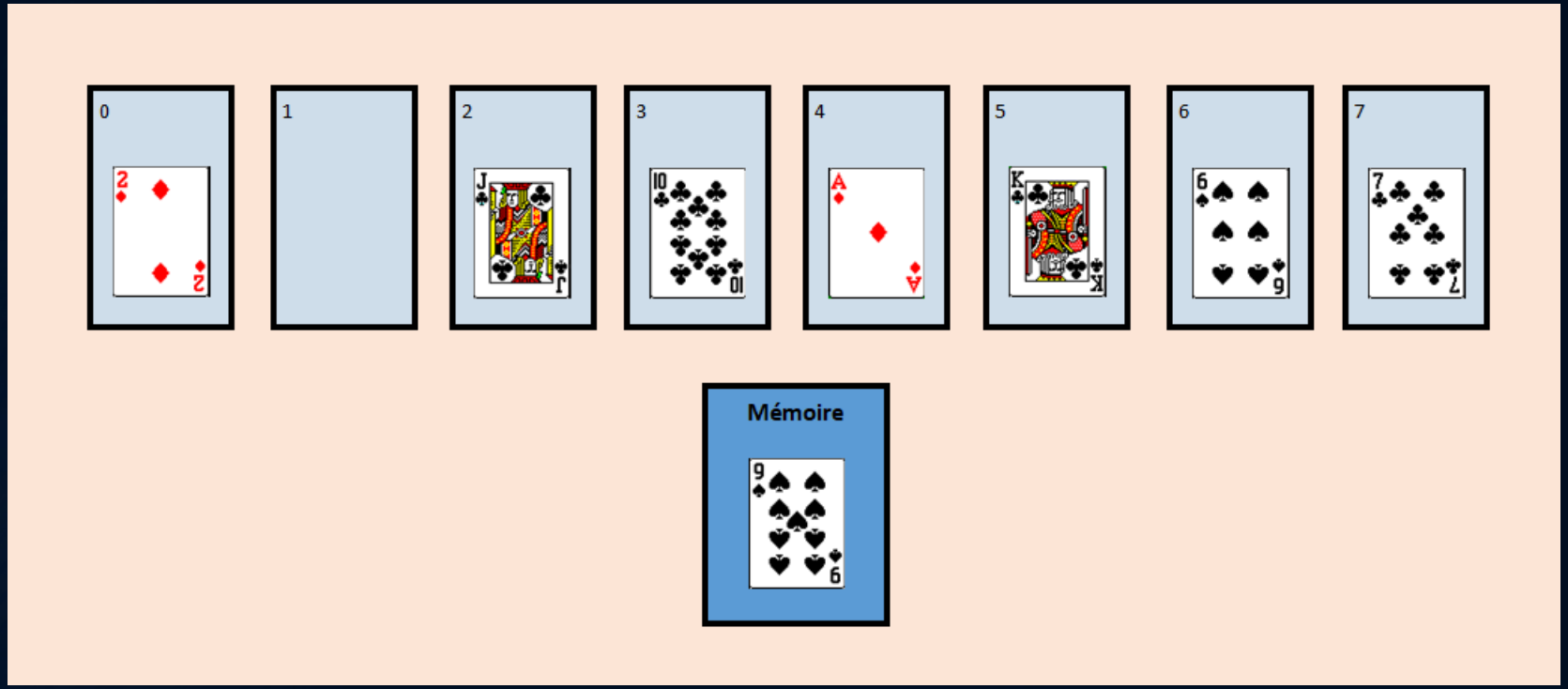
- Exemple d'application :



- 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 »

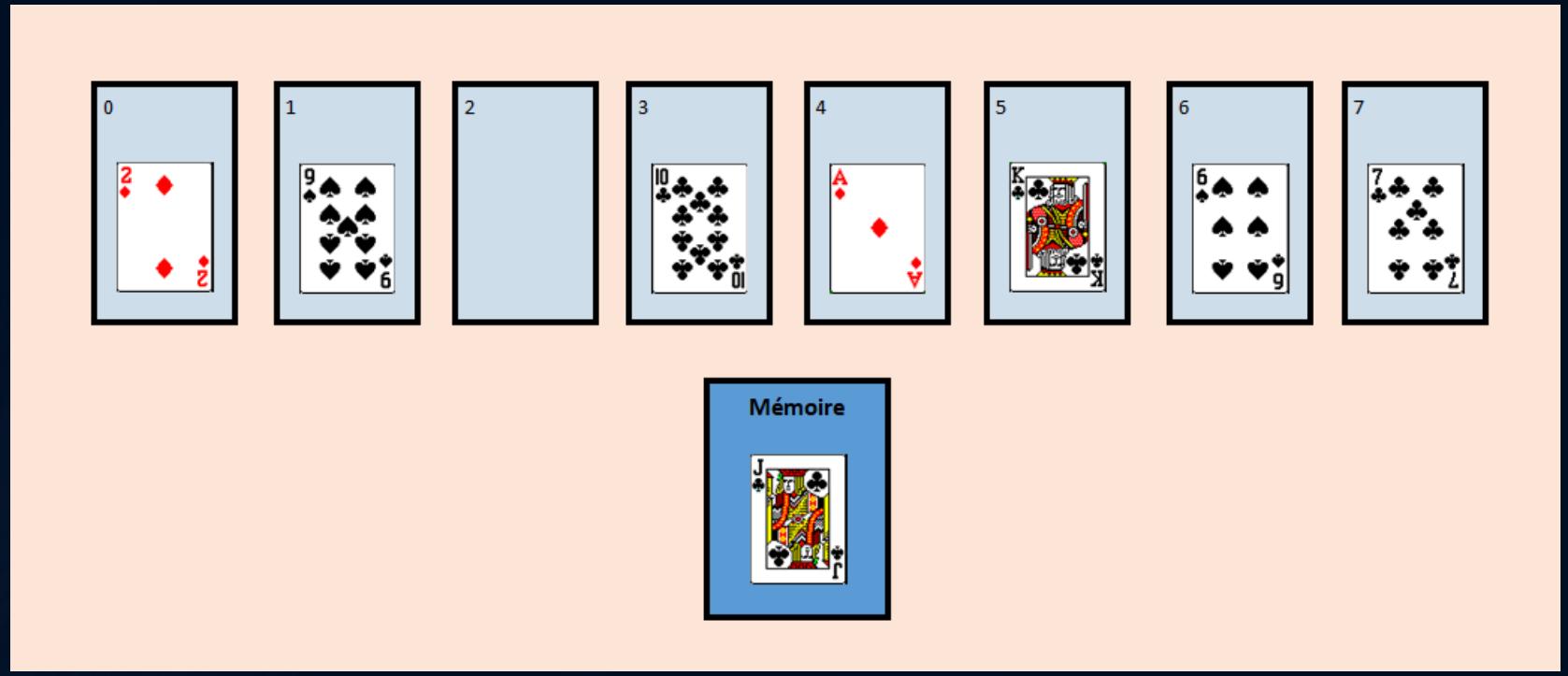
- Exemple d'application :



- 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 »

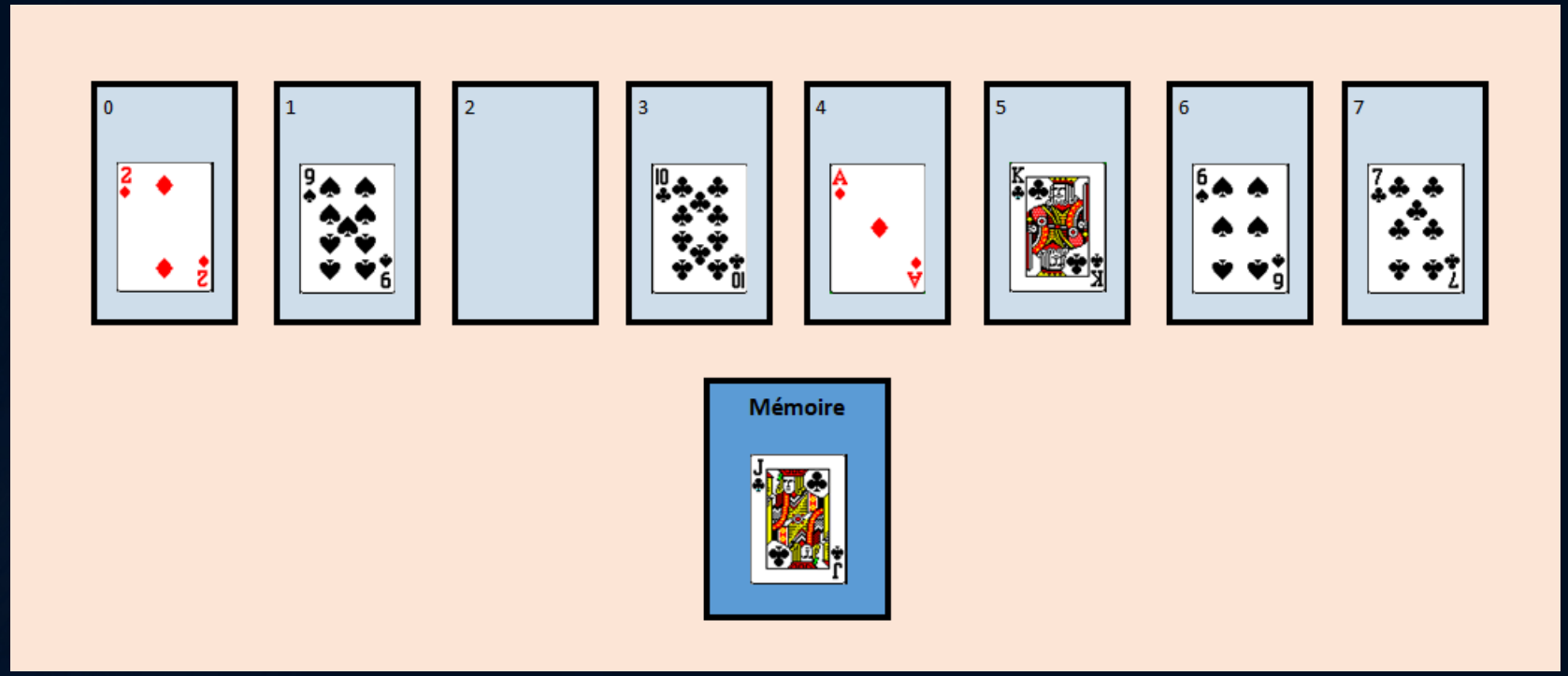
- Exemple d'application :



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

« Diviser pour régner »

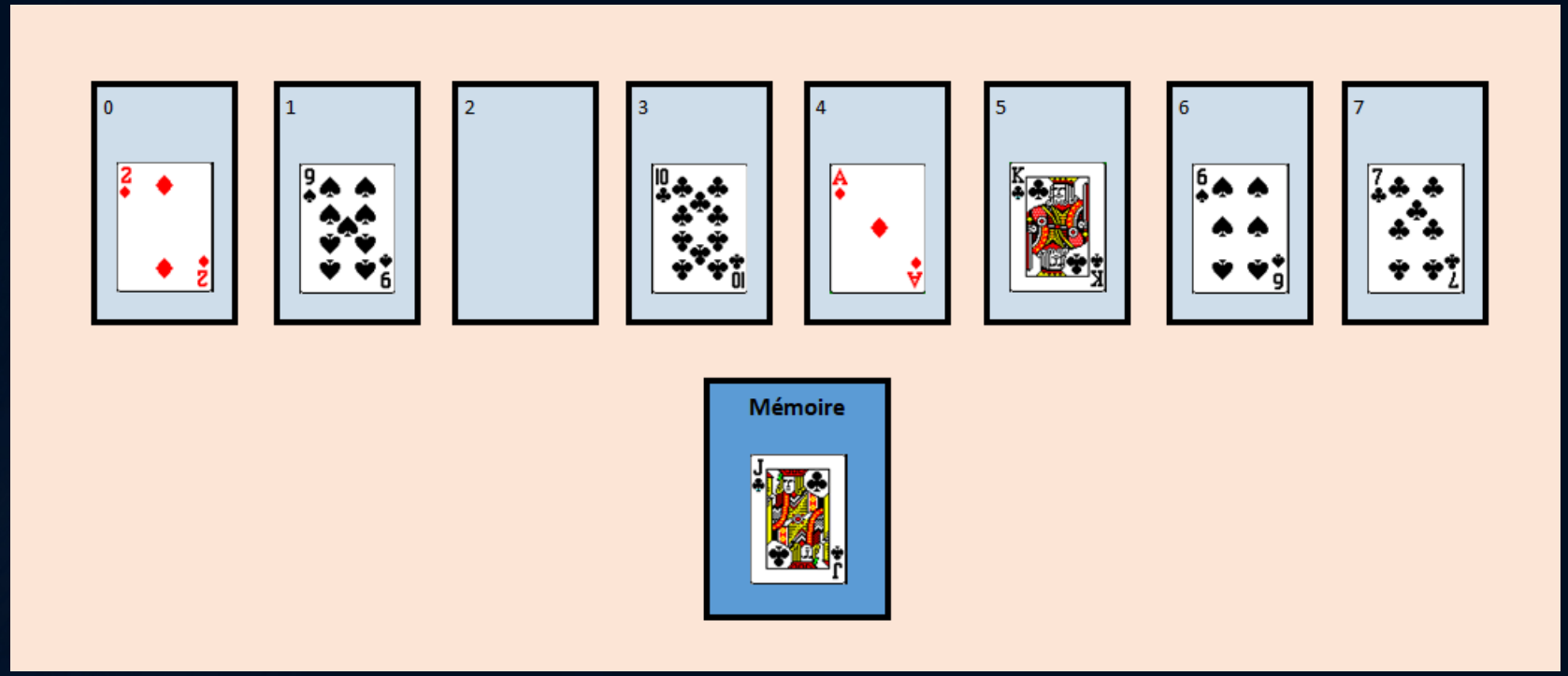
- Exemple d'application :



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

« Diviser pour régner »

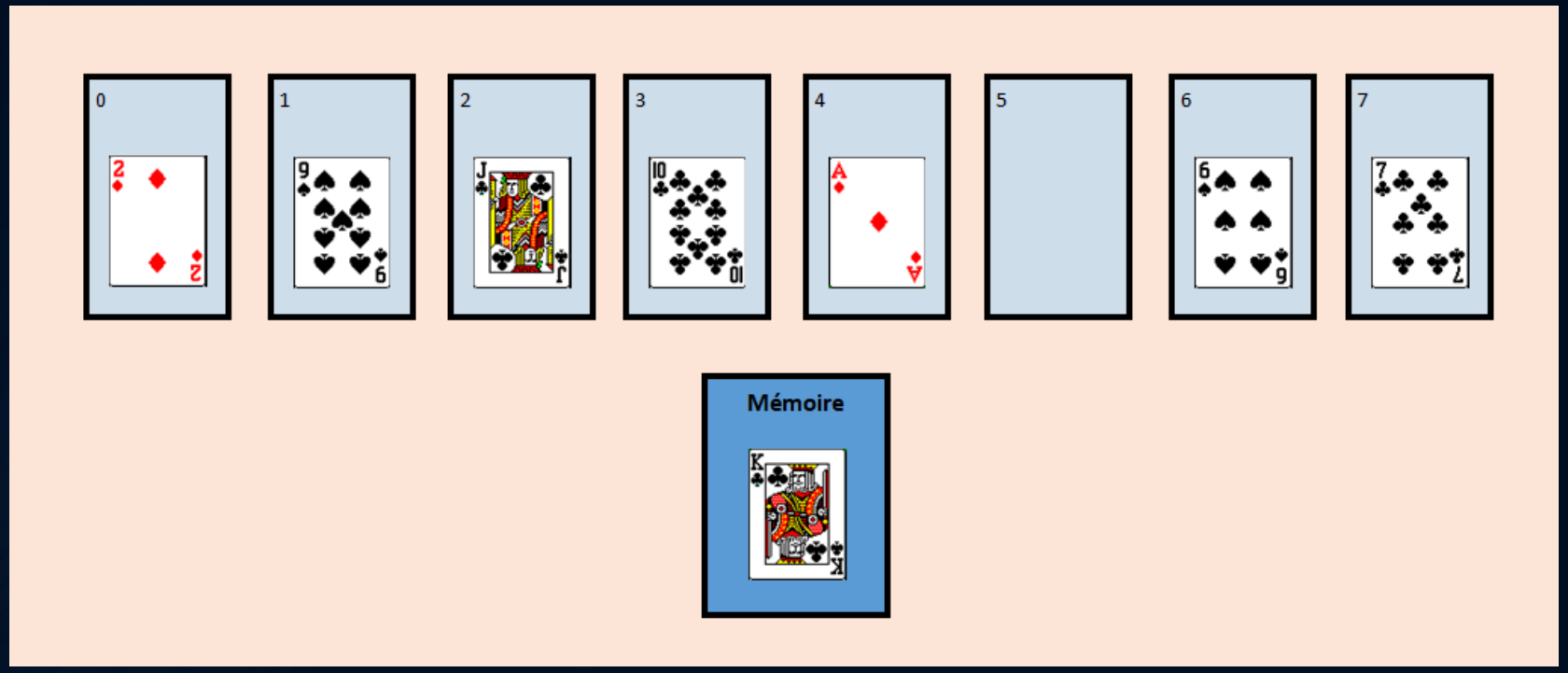
- Exemple d'application :



- 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 »

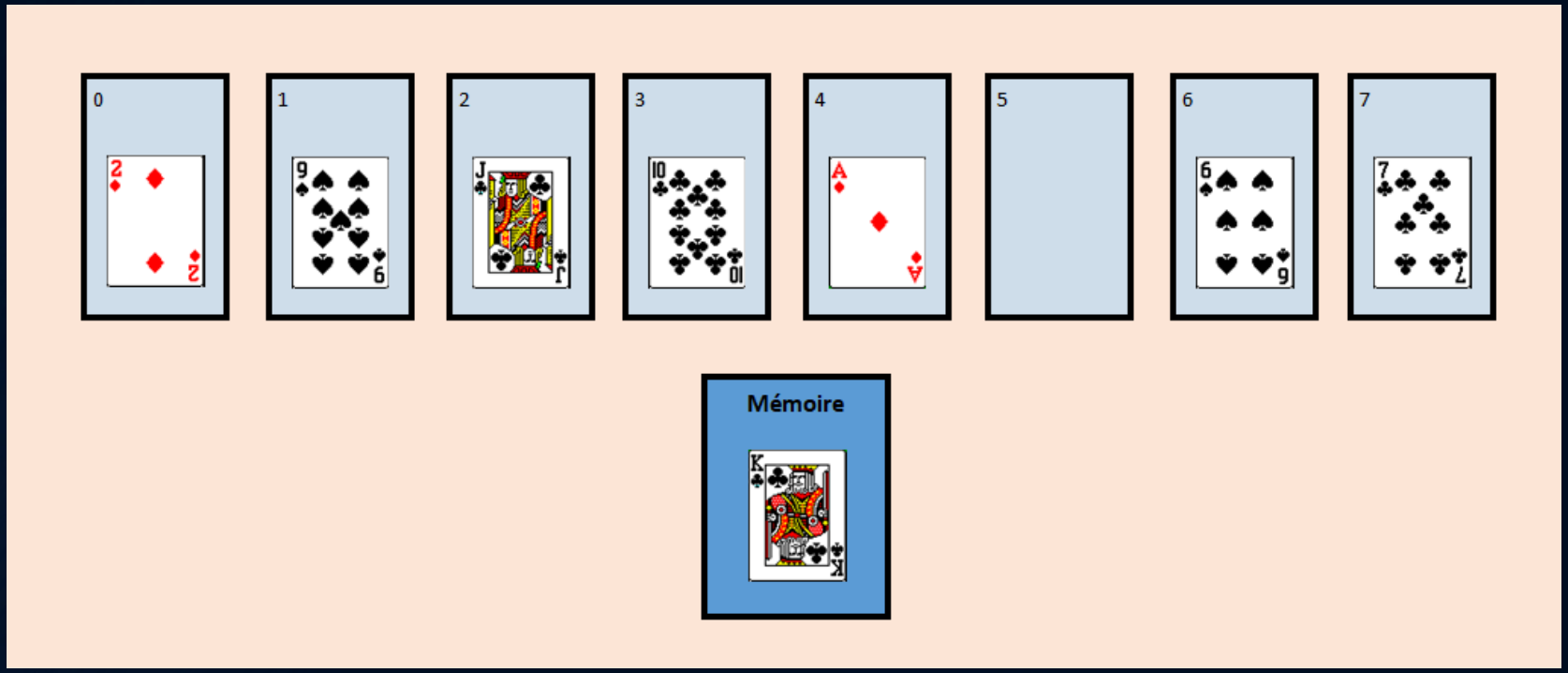
- Exemple d'application :



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

« Diviser pour régner »

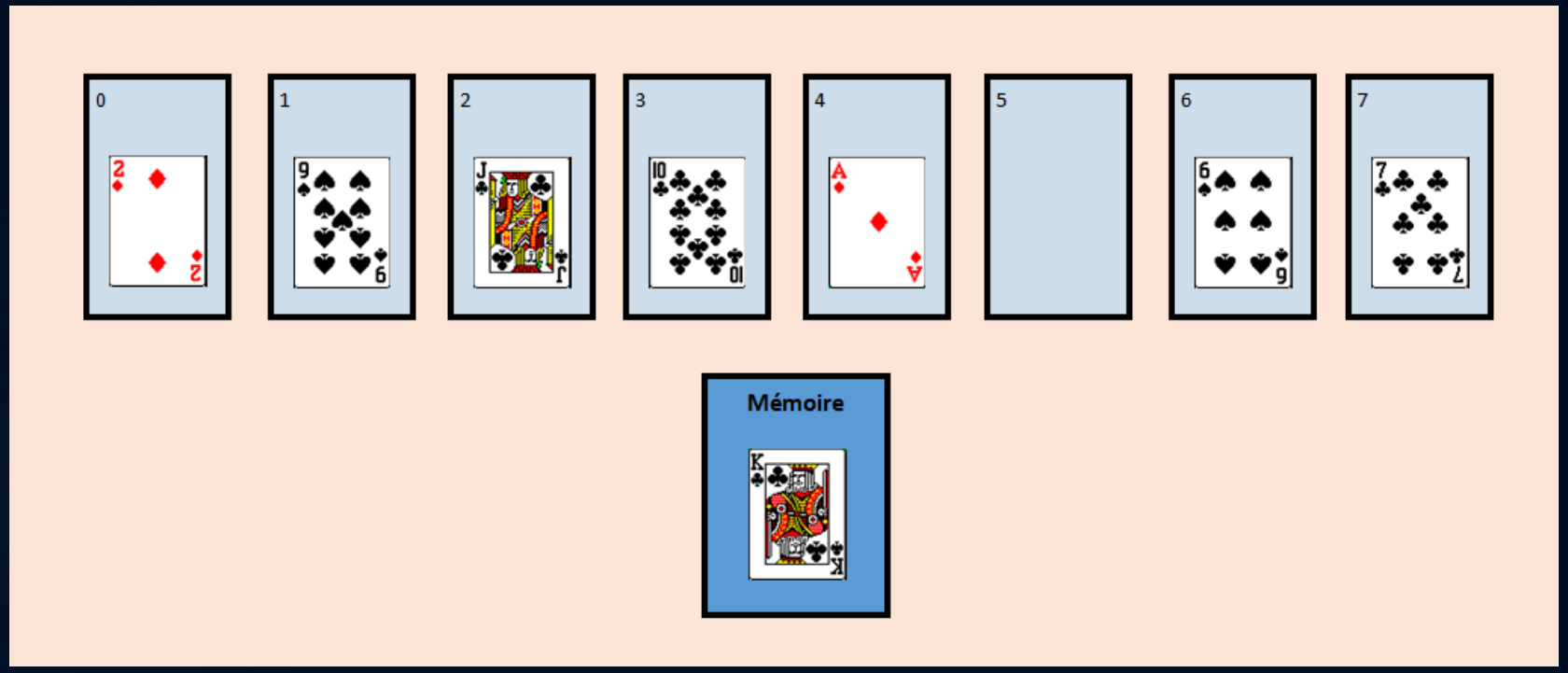
- Exemple d'application :



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

« Diviser pour régner »

- Exemple d'application :



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

« Diviser pour régner »

- Exercice 4 : Ecrire une fonction Python maximum **itérative** qui retourne l'élément maximal d'une liste d'au moins 1 entier positif.

« Diviser pour régner »

- Exercice 4 : 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 »

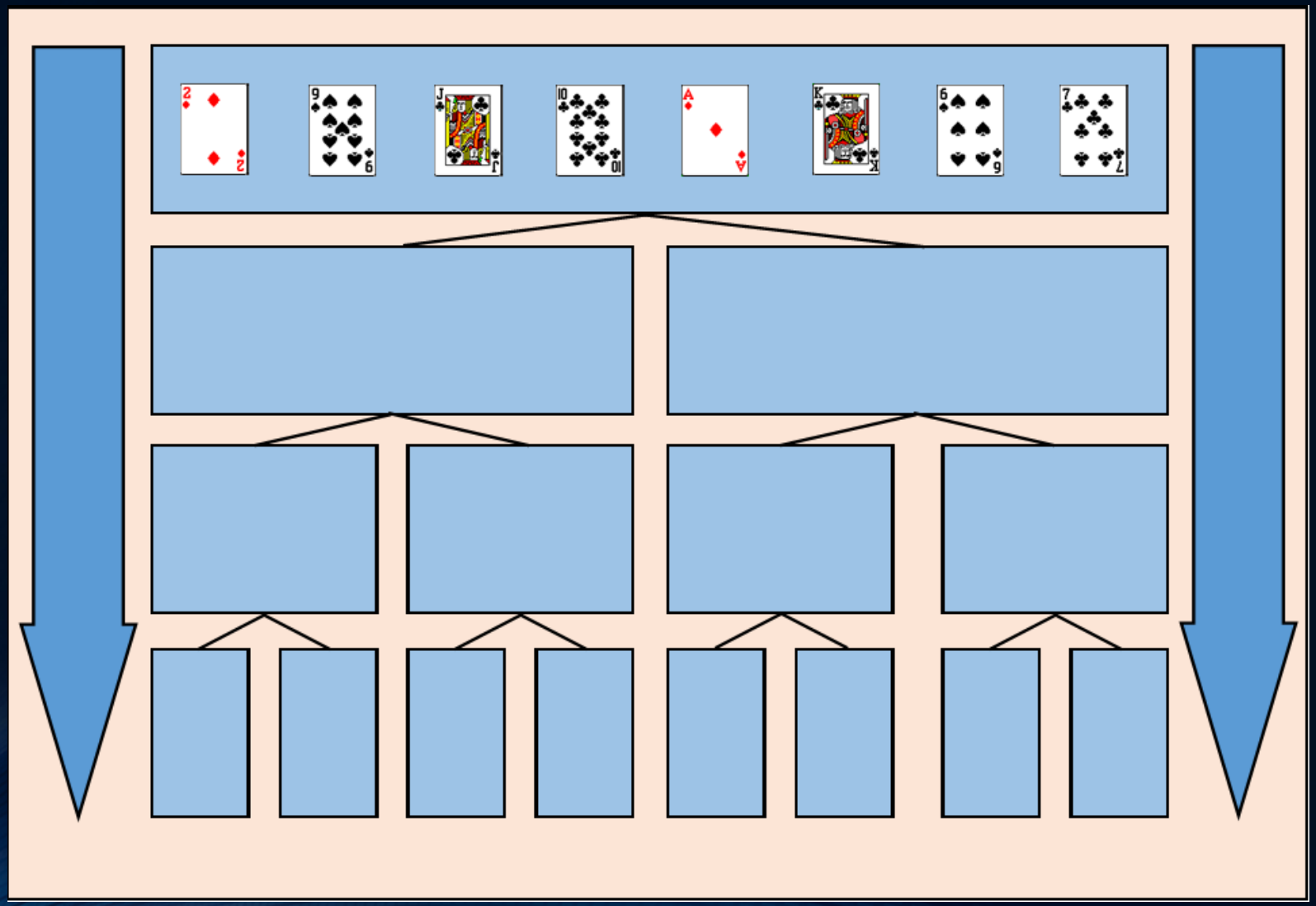
- Activité : On souhaite rechercher la carte de valeur maximale parmi une liste de cartes, mais cette fois-ci à l'aide de la méthode « **diviser pour régner** ».
- Choisissez 8 cartes au hasard.
- La lecture des cartes s'effectue cette fois-ci de **haut en bas** et de **bas en haut.**
- Adaptez la méthode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

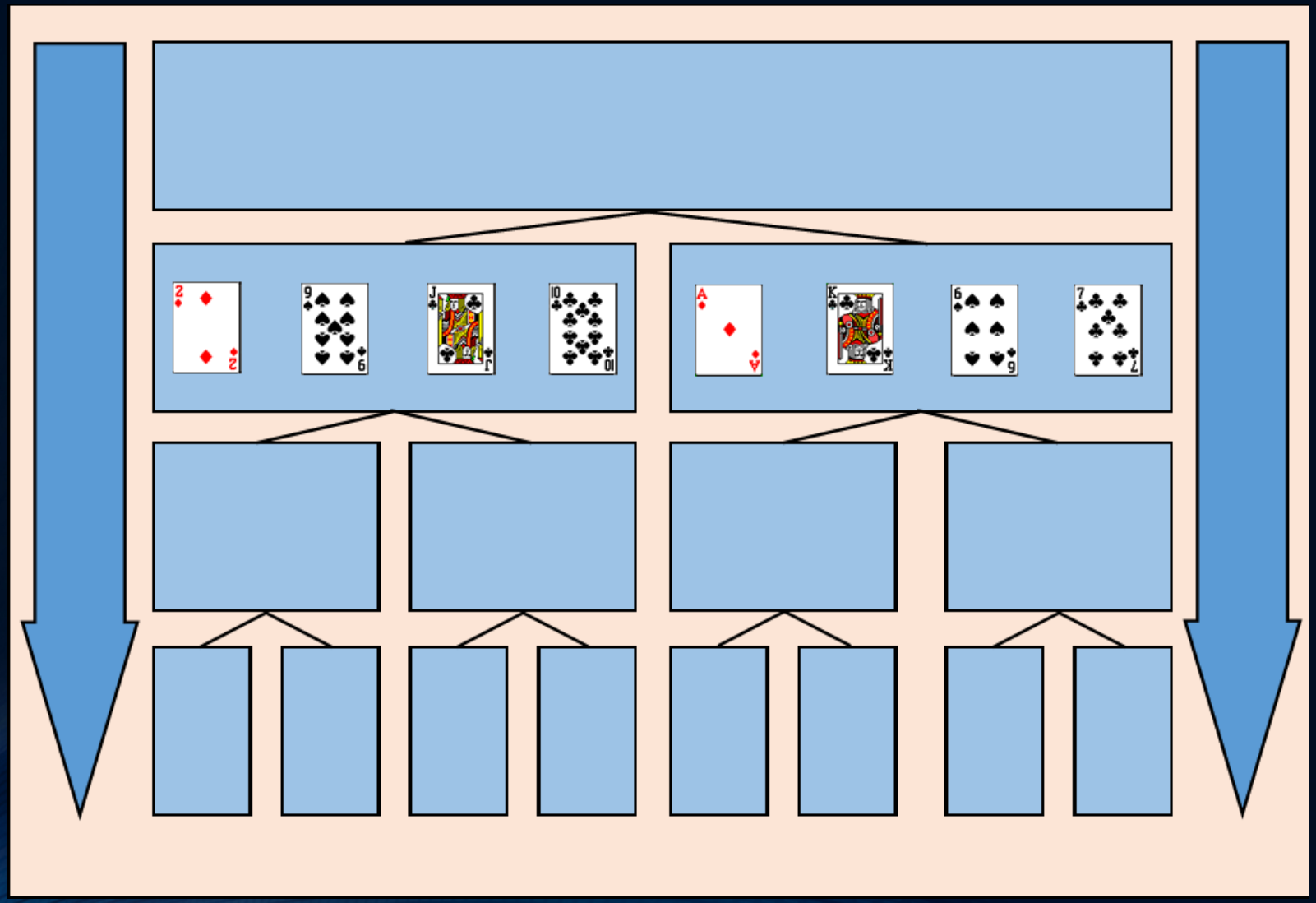
« Diviser pour régner »

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



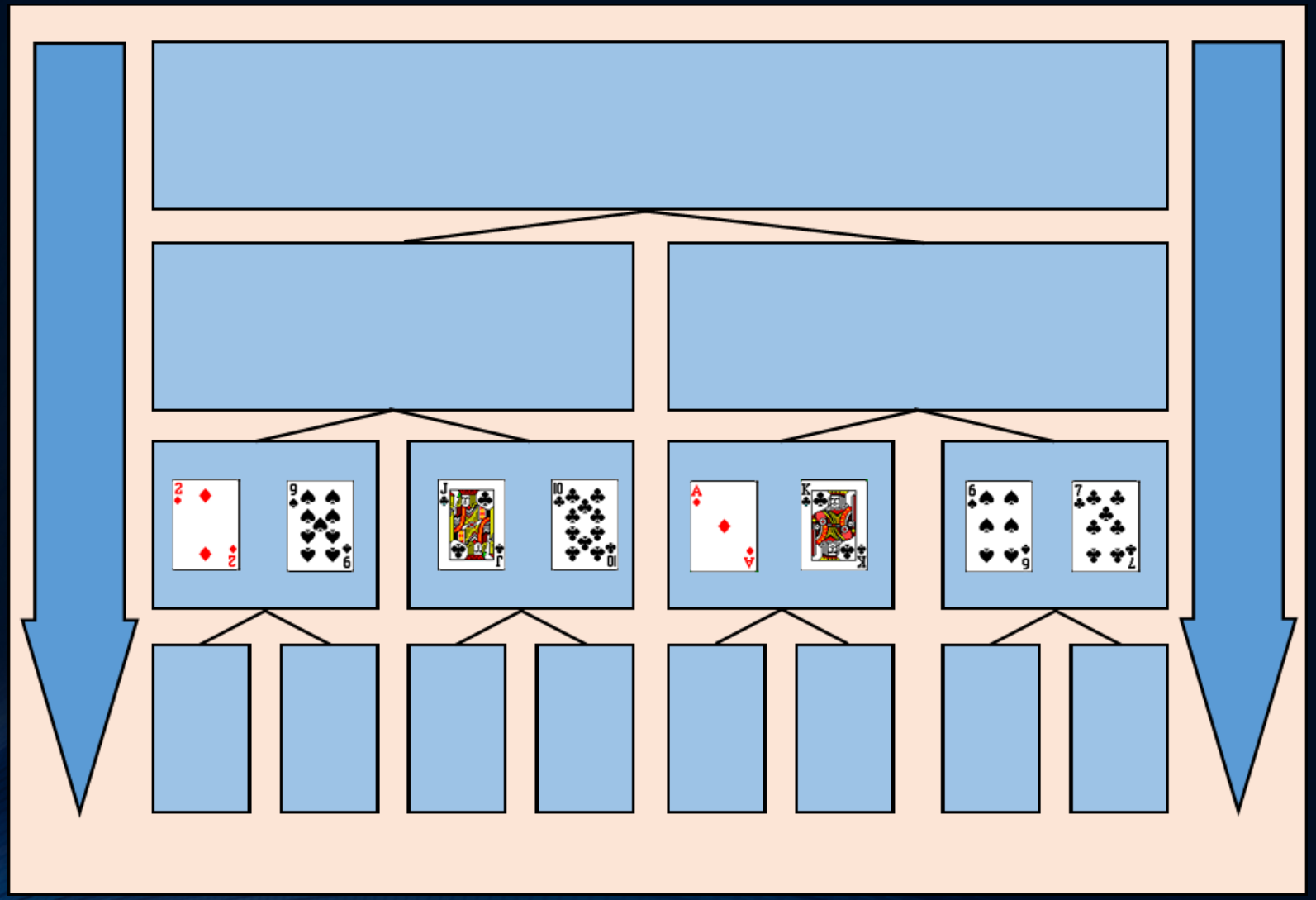
« Diviser pour régner »

- On divise de nouveau.



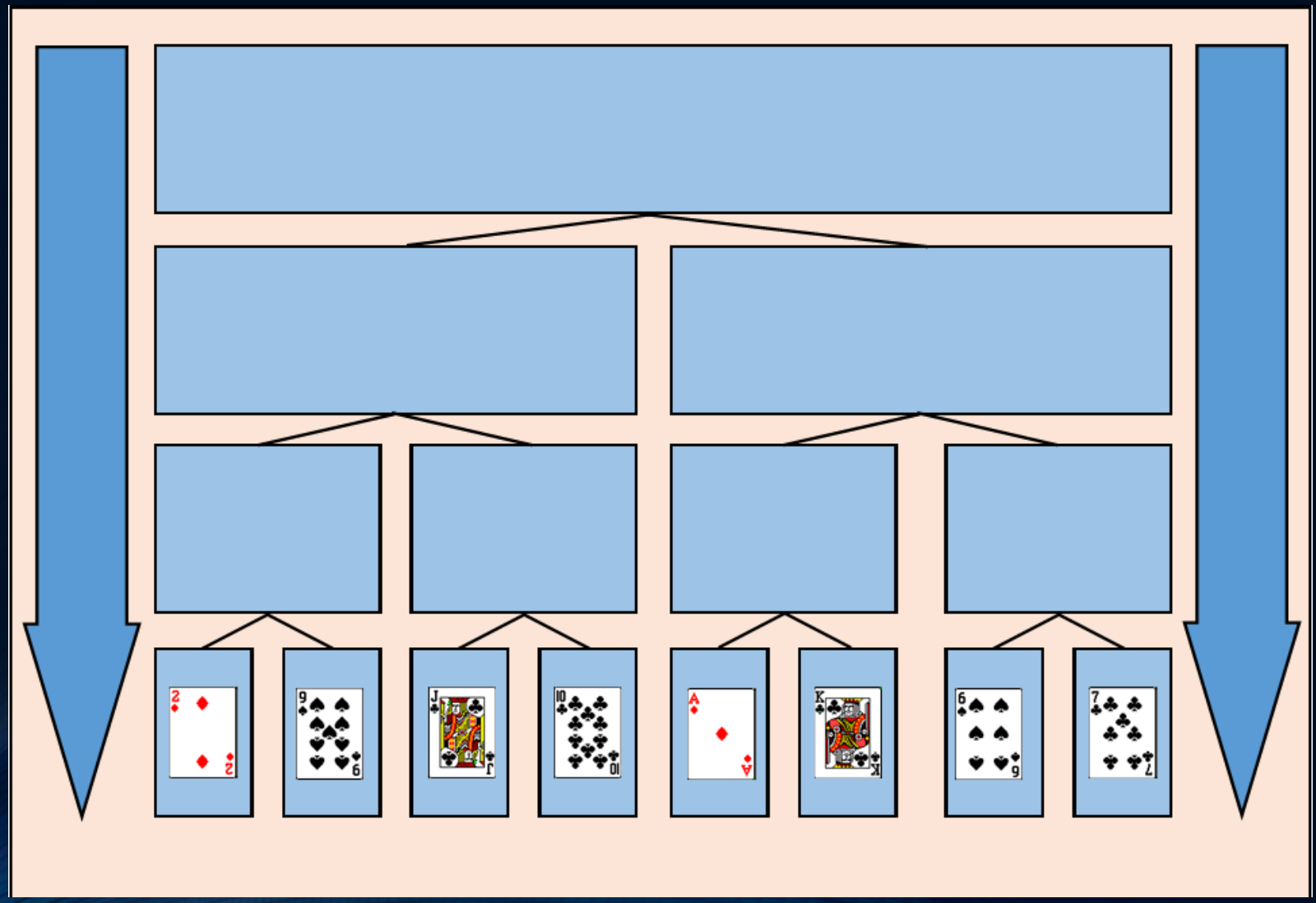
« Diviser pour régner »

- On divise de nouveau.



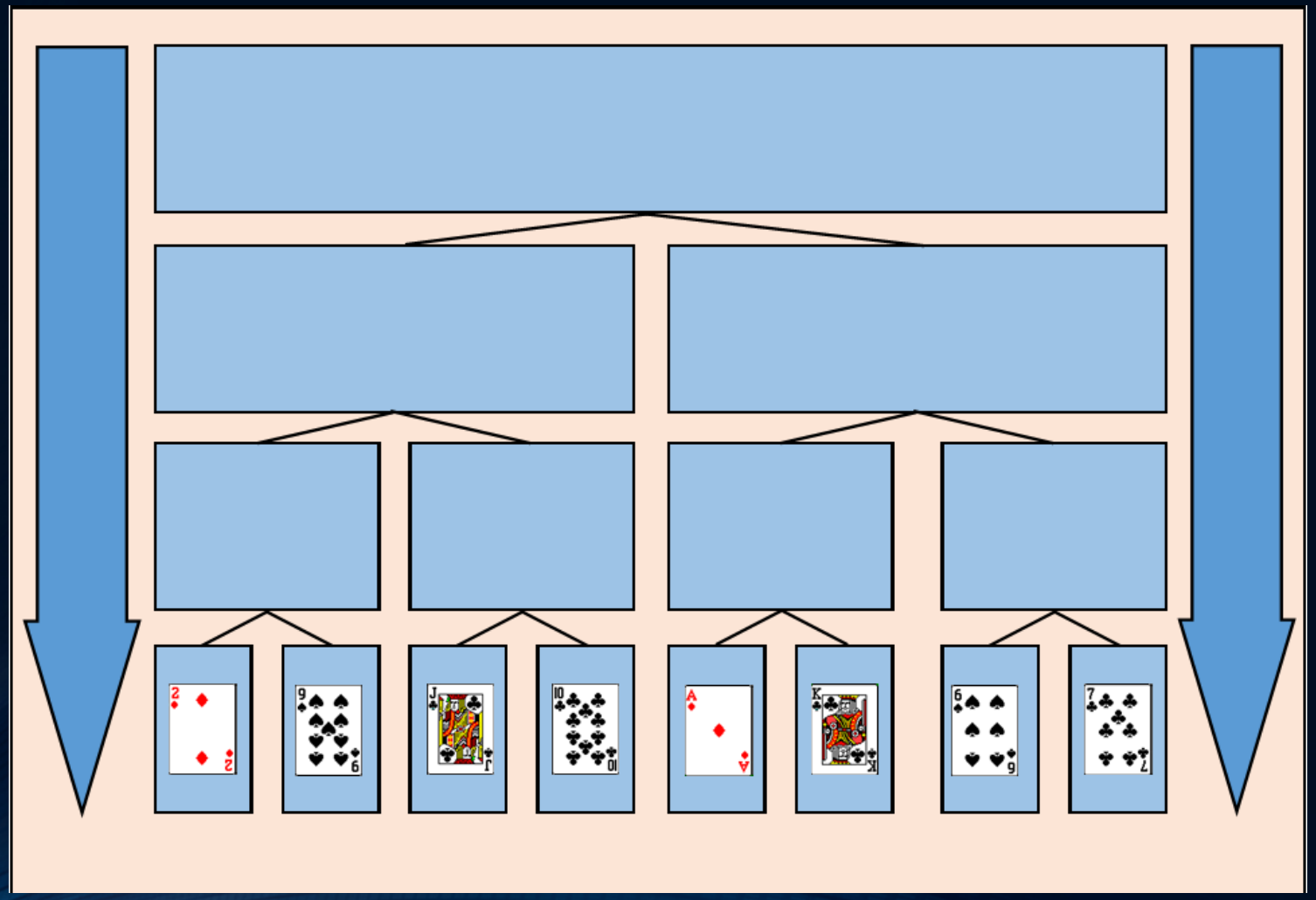
« Diviser pour régner »

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



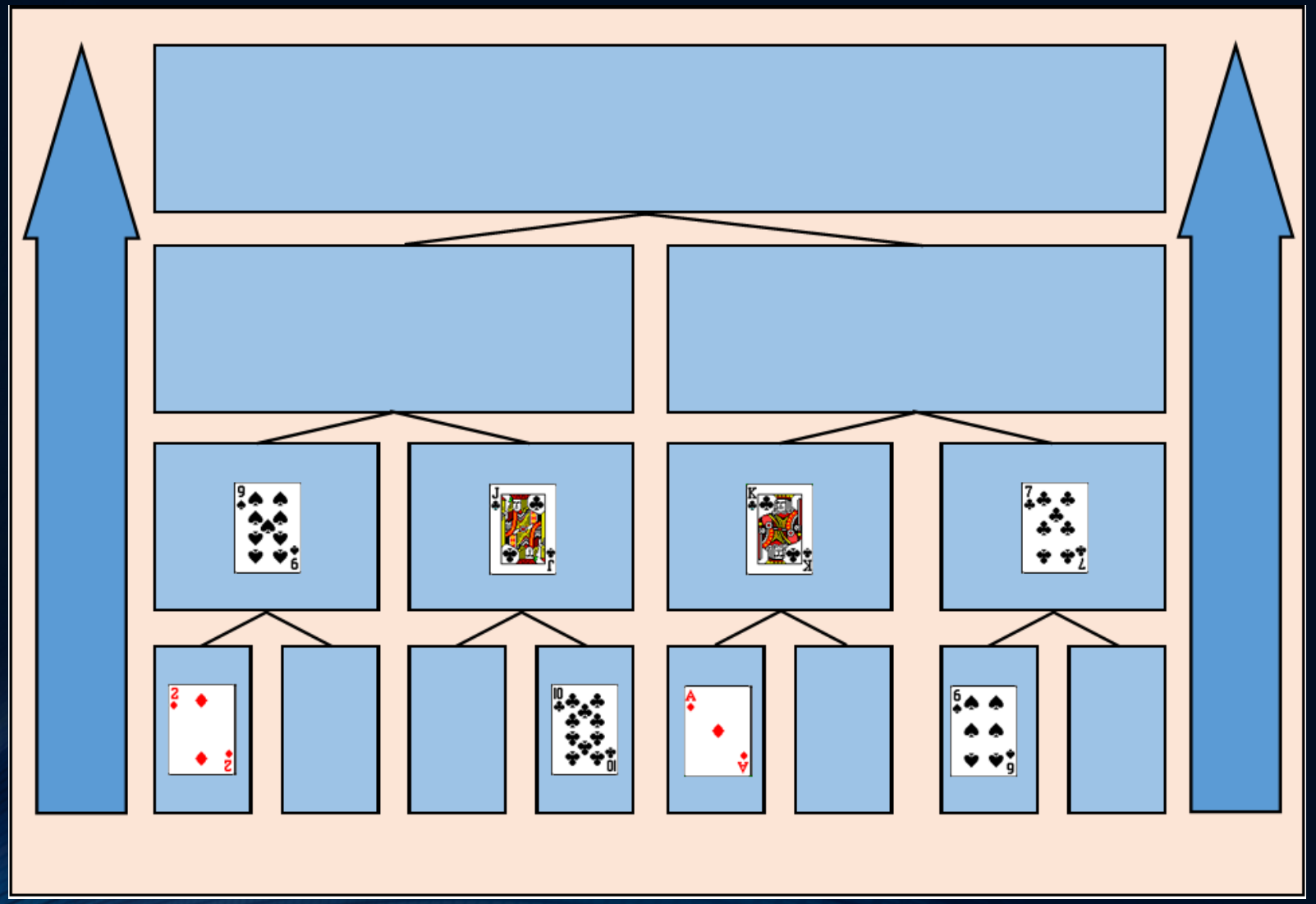
« Diviser pour régner »

- 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.



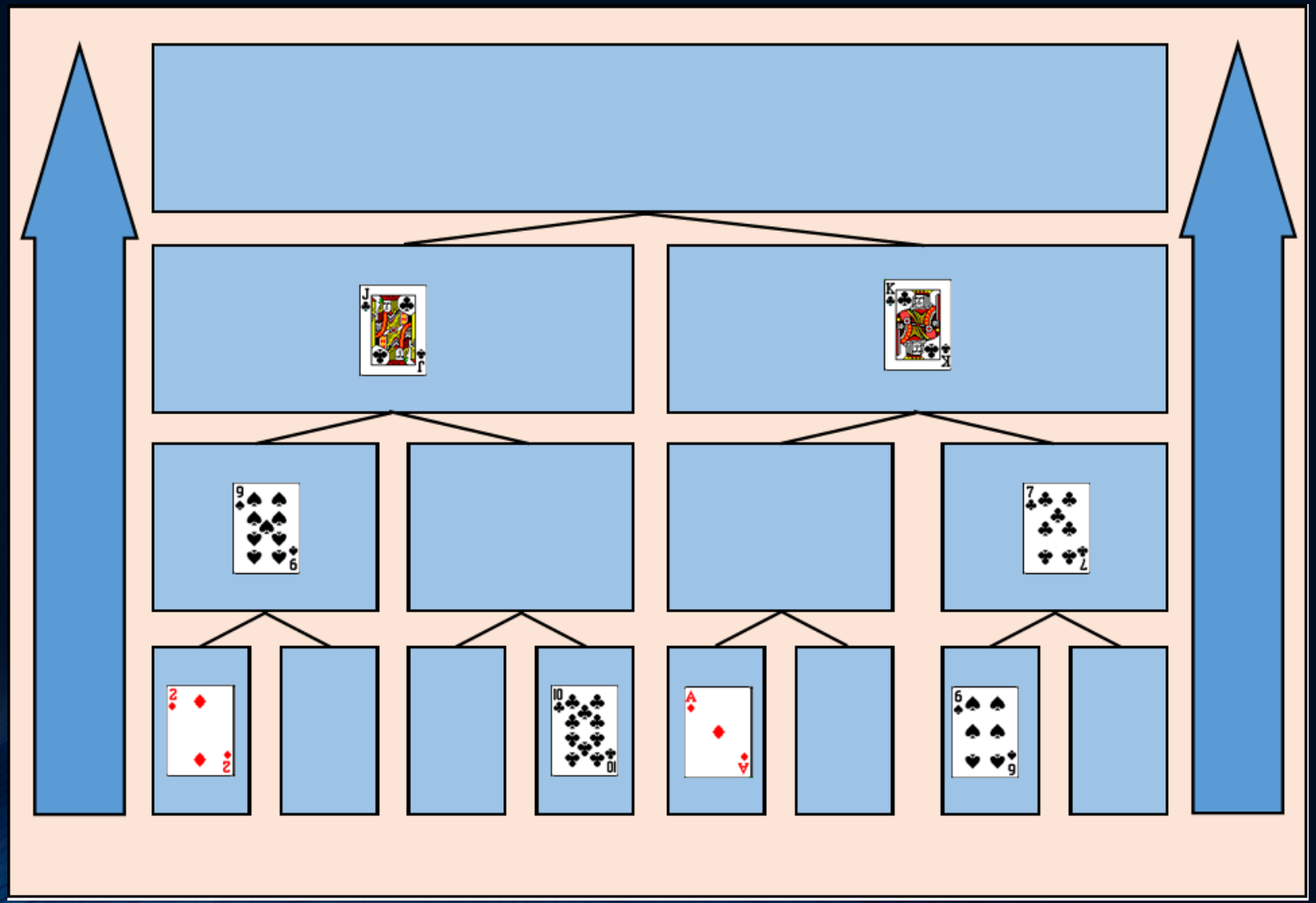
« Diviser pour régner »

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



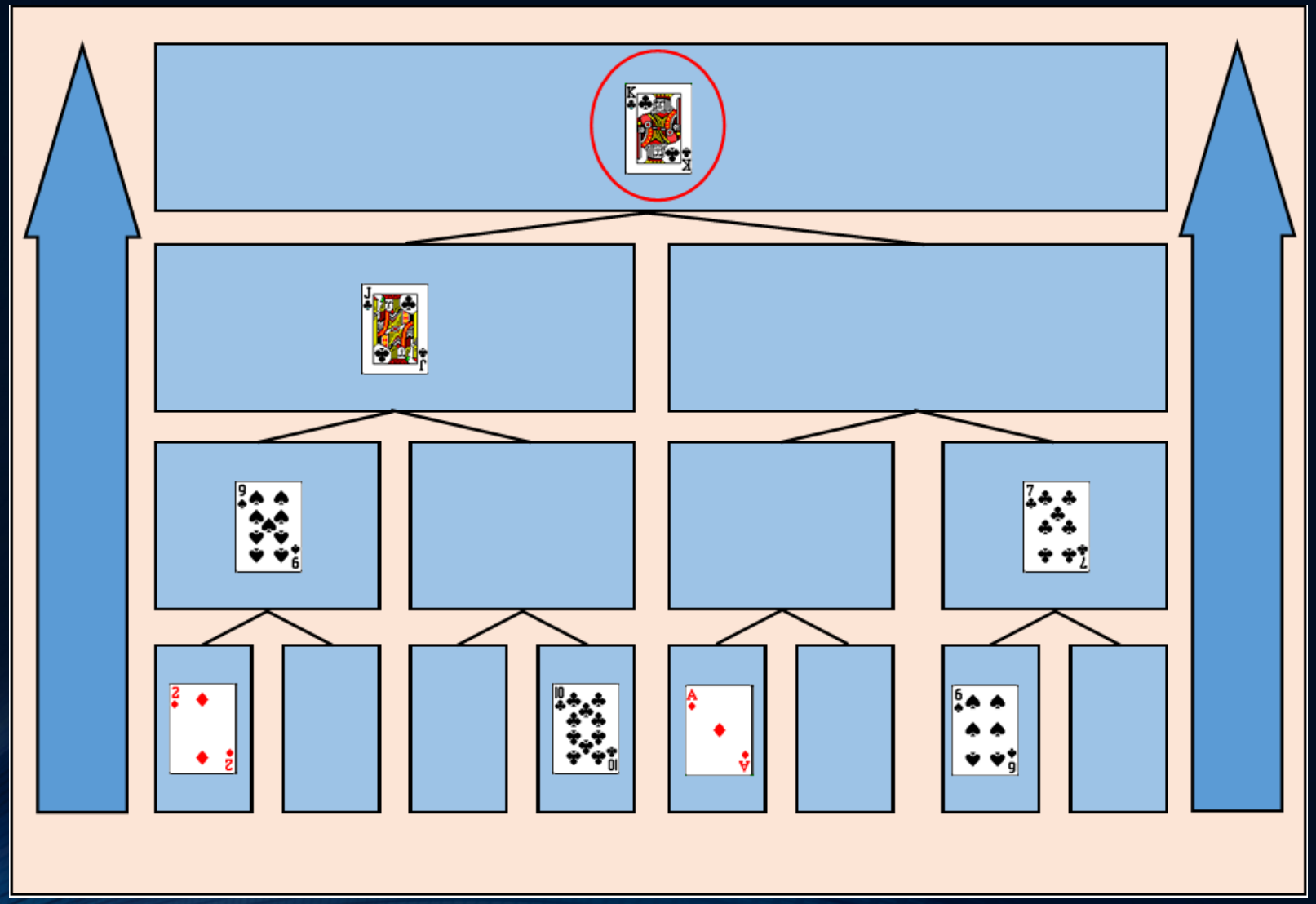
« Diviser pour régner »

- 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 »

- Exercice 5 : 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.
- 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 stocke la valeur max de la liste gauche dans une variable max_gauche.
  - On stocke la valeur max de la liste droite dans une variable max_droite.
  - Si max_gauche est supérieur à max_droite :
    - Retourner max_gauche
  - Sinon :
    - Retourner max_droite

« Diviser pour régner »

- Correction :

```
1  def maximum_r(liste):
2      if len(liste) == 1:
3          return liste[0]
4      else:
5          gauche, droite = liste[:len(liste)//2], liste[len(liste)//2:]
6          max_gauche, max_droite = maximum_r(gauche), maximum_r(droite)
7          if max_gauche > max_droite:
8              return max_gauche
9          else:
10             return max_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.

« Diviser pour régner »

- En première a été vue la recherche dichotomique, qui consiste à vérifier **dans un tableau trié** la présence d'un élément.
- On rappelle le fonctionnement de la recherche dichotomique :
  - On initialise une variable trouve à False.
  - On regarde l'élément central du tableau. S'il est égal à la clé :
    - trouve prend la valeur True
  - S'il est plus grand que la clé :
    - On recommence la recherche dans la partie du tableau allant du début à la valeur centrale.
  - Sinon :
    - On recommence la recherche dans la partie du tableau allant de la valeur centrale à la fin.

➢ Exercice 6 : Ecrire en Python l'algorithme de la recherche dichotomique de manière **itérative** puis **récursive**.

« Diviser pour régner »

- Exercice 6 : Ecrire en Python l'algorithme de la recherche dichotomique de manière **itérative**.
- Indications :
  - Deux variables debut et fin indiquant la partie de la liste sur laquelle on travaille.
    Au début, on travaille sur toute la liste.
  - Une variable milieu égale à l'indice de l'élément au milieu de la liste.
  - Tant que l'indice du début est inférieur ou égal à l'indice de fin :
    - SI l'élément est à l'indice milieu : On a trouvé l'élément !
    - SINON :
      - Si l'élément cherché est plus petit que l'élément du milieu : On recommence la recherche dans la partie de gauche. (Modifier les variables debut et fin en conséquence).
      - Sinon : On recommence la recherche dans la partie de droite.
        (Modifier les variables debut et fin en conséquence).

« Diviser pour régner »

- Exercice 6 : Ecrire en Python l'algorithme de la recherche dichotomique de manière **itérative**.

```python
def dicho_i(liste, elt):
    """ Algorithme itératif de recherche d'un élément dans une liste triée.
    :param liste: (list d'int) Liste triée
    :param elt: (int) Element à chercher
    :CU: len(liste) >= 1 """

    deb = 0
    fin = len(liste) - 1
    m = (deb + fin) // 2
    while deb <= fin :
        if liste[m] == elt :
            # L'élément est à l'indice m
            return True
        elif liste[m] > elt :
            # On regarde avant l'indice m
            fin = m - 1
        else :
            # On regarde après l'indice m
            deb = m + 1
        m = (deb + fin) // 2
    return False
```

« Diviser pour régner »

- Exercice 6 : Ecrire en Python l'algorithme de la recherche dichotomique de manière **récursive**.
- Indications :
  - Prend en entrée 4 paramètres : La liste, l'élément recherché, l'indice de gauche et l'indice de droite (délimitant la partie de la liste traitée). Définir g et d à -1 par défaut (par exemple).
  - Si g et d sont à -1, g prend la valeur 0 et d prend la valeur du dernier indice de liste.
  - Si g > d, on retourne False.
  - Une variable m prend la valeur (g + d) // 2 (c-à-d le milieu de la zone de recherche)
  - Si l'élément à l'indice m est égal à l'élément recherché :
    - Retourner True.
  - Sinon si l'élément à la position m est supérieur à l'élément recherché :
    - Retourner un appel récursif à la même fonction, en changeant les valeurs de g et d.
  - Sinon:
    - Retourner un appel récursif à la même fonction, en changeant les valeurs de g et d.

« Diviser pour régner »

- Exercice 6 : Ecrire en Python l'algorithme de la recherche dichotomique de manière **récursive**.

```python
def dicho_r(liste, elt, g=None, d=None):
    """ Algorithme récursif de recherche d'un élément dans une liste triée.
    :param liste: (list d'int) Liste triée
    :param elt: (int) Element à chercher
    :param g: (int) Premier élément pris en compte dans la liste
    :param d: (int) Dernier élément pris en compte dans la liste
    :CU: len(liste) >= 1
    :Conditions d'arrêt: Si g > d, ou si l'élément est trouvé. """

    # Si g et d ne sont pas définis, on les définit :
    if g == d == None:
        g = 0
        d = len(liste) - 1

    # Si l'indice de gauche est supérieur à l'indice de droite
    if g > d:
        return False

    m = (g + d) // 2 # Définir le milieu
    if liste[m] == elt:
        return True
    elif elt < liste[m]:
        return dicho_r(liste, elt, g, m - 1)
    else:
        return dicho_r(liste, elt, m + 1, d)
```

« Diviser pour régner »

- On souhaite comparer la méthode de recherche par **dichotomie** avec la recherche par **balayage**.
- On rappelle l'algorithme de recherche par balayage :

```
1  def recherche(liste, elt):
2      for i in range(len(liste)):
3          if liste[i] == elt:
4              return True
5      return False
```

➢ A l'aide de la fonction process_time du module time, on peut mesurer le temps d'exécution des deux méthodes de recherche afin d'essayer d'en mesurer la complexité.

« Diviser pour régner »

- A l'aide de la fonction process_time du module time, mesurer le temps d'exécution des deux méthodes de recherche (dichotomie et recherche par balayage), en faisant varier la taille de la liste et en recherchant le dernier élément de la liste. (Pire des cas)

  Utiliser le fichier comparer_dicho_balayage.py du dossier Code du Github et exécutez le code en faisant varier le nombre de valeurs.

Nombre de valeurs	Temps dichotomie (en s)	Temps balayage (en s)
1 000 000		
2 000 000		
4 000 000		
8 000 000		
16 000 000		
32 000 000		

« Diviser pour régner »

- Voici le tableau rempli, à noter que le temps d'exécution varie d'un ordinateur à l'autre (car pas le même matériel) :

Nombre de valeurs	Temps dichotomie (en s)	Temps balayage (en s)
1 000 000	0,0	0,0625
2 000 000	0,0	0,109375
4 000 000	0,0	0,21875
8 000 000	0,0	0,46875
16 000 000	0,0	0,890625
32 000 000	0,0	2,046875

- Cette mesure permet de voir que la **méthode dichotomique** est nettement plus efficace. Elle semble être de complexité logarithmique, là où la **méthode par balayage** semble être de complexité linéaire (car le temps d'exécution double chaque fois que l'on double le nombre n de valeurs).

« Diviser pour régner »

- Validons les mesures par la méthode théorique (dans le cas où l'élément cherché est le dernier de la liste) :

n =	Dichotomie (récursif)	Balayage
5	4	3
10	5	6
20	7	11
35	10	18
40	9	21
80	11	41

- Combien y a-t-il de comparaisons en fonction de n pour la méthode par balayage ?

« Diviser pour régner »

- Validons les mesures par la méthode théorique :

n =	Dichotomie (récursif)	Balayage
5	4	3
10	5	6
20	7	11
35	10	18
40	9	21
80	11	41

- Pour la méthode par balayage, dans le pire cas :
  - comp(n) = $\frac{n+2}{2}$ si n est pair
  - comp(n) = $\frac{n+1}{2}$ si n est impair

La complexité est donc d'ordre O(n), elle est donc bien linéaire.

« Diviser pour régner »

n =	Dichotomie (récursif)	Balayage
5	4	3
10	5	6
20	7	11
35	10	18
40	9	21
80	11	41
160	13	81

- Pour la méthode par dichotomie :
  - Chaque fois qu'on double n, le nombre de comparaison augmente de 2, on peut donc écrire la complexité :

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

  - Cela confirme que la méthode dichotomique est de complexité logarithmique d'ordre O(log(n)), avec un nombre de comparaisons : $comp(n) \cong log_{1,45}(n)$

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 :

```
1  def tri_insertion(tab):
2      taille = len(tab)
3      for j in range(1, taille):
4          cle = tab[j]
5          i = j - 1
6          while i >= 0 and tab[i] > cle:
7              tab[i + 1] = tab[i]
8              i = i - 1
9          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 :

```
1  def tri_selection(tab):
2      # Pour chaque valeur du tableau :
3      for i in range(len(tab)):
4
5          # Trouver le minimum parmi les valeurs suivantes
6          min = i
7          for j in range(i+1, len(tab)):
8              if tab[min] > tab[j]:
9                  min = j
10
11         # Echanger la valeur à l'indice i avec la valeur min
12         tmp = tab[i]
13         tab[i] = tab[min]
14         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 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.

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 :

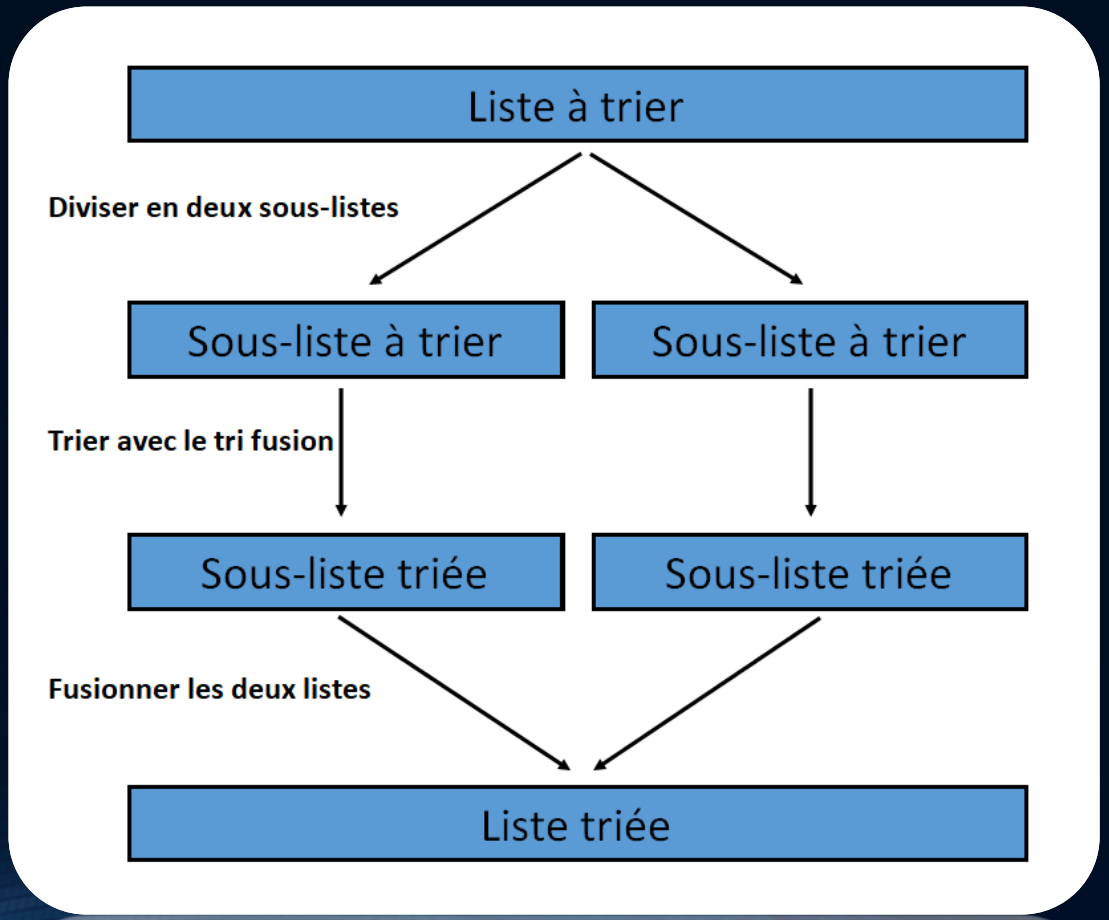
Ecrire maintenant la fonction Python **récursive** fusion qui retourne la **fusion** de **deux listes triées**.

```
1  def fusion(L1, L2):
2      """ Fonction retournant une liste triée, fusion de deux listes triées.
3      :param L1: (list d'int) Liste d'entiers triée
4      :param L2: (list d'int) Liste d'entiers triée
5      :return: (list d'int) Fusion des deux listes triées """
6
7      if L1 == []:
8          return L2
9      elif L2 == []:
10         return L1
11     elif L1[0] < L2[0]:
12         return [L1[0]] + fusion(L1[1:], L2)
13     else:
14         return [L2[0]] + fusion(L1, L2[1:])
```

Vers de nouvelles méthodes de tri

- Exercice 7 suite :

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**.

```
1  def tri_fusion(Liste):
2      """ Effectue le tri fusion de la liste passée en entrée.
3      :param Liste: (list d'int) Liste d'entiers
4      :return: (list d'int) Une liste d'entiers triée """
5
6      n = len(Liste)
7      if n <= 1:
8          return Liste
9      else:
10         m = n // 2
11         return fusion(tri_fusion(Liste[:m]), tri_fusion(Liste[m:]))
```

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 :

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

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

  n – 1 et n – 2 sont d'ordre linéaire, on peut donc approximer la complexité :

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

  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