Schéma de récursivité
Méthode récursive
Analyse asymptotique d'un algorithme récursif
Récursion de queue

Schéma de récursivité

- **Récursivité** : quand une méthode s'appelle elle-même
- L'exemple classique est la fonction factorielle :

  n! = 1· 2· 3· ··· · (n-1)· n
- Définition récursive :

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

```
def f( n ):
   if n == 0:
      return 1
   else:
      return n * f( n - 1 )
```

Contenu d'une méthode récursive

- Cas de base
  - Valeurs du ou des arguments pour lesquelles il n'y a pas d'appel récursif (il doit y avoir au moins un cas de base)
  - Toute chaîne possible d'appels récursifs doit éventuellement atteindre un cas de base
- Appels récursifs
  - Appels à la méthode courante
  - Appels qui doivent être définis de manière à progresser vers un cas de base (exemple : décrément de n dans l'énoncé `return n * f( n - 1 )`

```python
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "7 janvier 2014"
#
# Programme Python pour IFT2015/Mise à niveau/Initiation à Python
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

# Ce programme prend en input une valeur entière
# et retourne en output la factorielle de cette valeur.

# utilise sys pour saisir les arguments de la ligne de commande
import sys

# utilise time pour mesure le temps d'exécution
import time

# utilise optparse pour parser la ligne de commande
from optparse import OptionParser

# Usage: factorielle.py [options]
#
# Options:
#   -h, --help      show this help message and exit
#   -n N, --n=N     entier positif
#   -v, --verbose   trace les appels récursifs de la fonction factorielle

# Fonction principale
def main( argv ):

    # variables globales pour les options et arguments
    global opts
    global args

    # mettre les options et arguments
    parser = OptionParser()
    parser.add_option( "-n", "--n", dest = "n", default = 0,
                       help = "entier positif", metavar = "N" )
    parser.add_option( "-v", "--verbose",
                       action = "store_true", dest = "verbose", default = False,
                       help = "trace les appels récursifs de la fonction factorielle" )

    # parse the options and arguments
    opts, args = parser.parse_args()

    # s'assurer que n est un entier positif
    try:
        n = int( opts.n )
    except ValueError:
        print( opts.n, "n'est pas un entier !" )
        exit()

    # ici n est un entier
    # on s'assure qu'il est positif
    if n < 0:
        print( n, "doit être un entier positif !" )
        exit()
    else:

        # Calculer la factorielle de cet entier et
        # sauvegarder le résultat dans une variable locale.
        # On peut activer ou non la trace d'exécution de
        # la fonction en 2è argument qui par défaut est False.

        avant = time.time()
        fact = factorielle( n, opts.verbose )
        apres = time.time()

        # Afficher le résultat
        print( 'La factorielle de', n, 'est', fact, 'calculée en', apres - avant, 'secondes' )
```

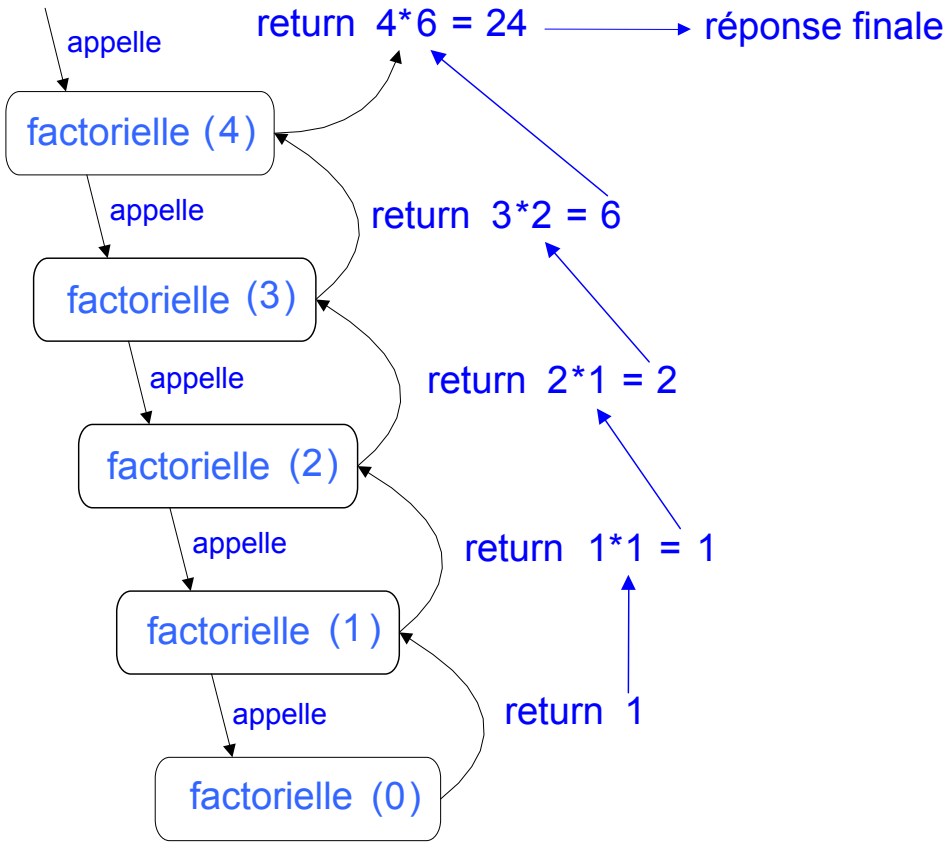
```
# Fonction factorielle d'un entier positif n:
#  n! = 1, si n = 0; n.(n-1).(n-2). ... 3.2.1 si n >= 1
#    donne le nombre de permutations de n objets distincts.
#    Par exemple, on peut permutter les trois caractères x, y et z
#    de 3! = 3.2.1 = 6 manières différentes: xyz, xzy, yxz, yzx,
#    zxy et zyx.
#    La fonction possède une definition récursive naturelle, par
#    exemple 13! = 13.12!, et n! = 1 si n = 0; n.(n-1)! si n >= 1.
#    0! représente le cas de base qui n'est pas definit recursivement,
#    le f( n-1 ) dans n x f( n-1 ) représente le cas récursif.
#    Trace d'exécution possible avec le 2è argument par défaut à False.
#    La profondeur d'exécution est initialisée à 0 et on l'utilise
#    pour indenter l'affichage de l'appel de la fonction.

def factorielle( n, trace = False, profondeur = 0 ):
    if n == 0:
        if( trace ):
            print( profondeur * ' ', 'return 1' )
        return 1
    else:
        if( trace ):
            print( profondeur * ' ', 'return ', n, '* factorielle(', n - 1, ')' )
        return n * factorielle( n-1, trace, profondeur+1 )

#    La fonction n'utilise pas d'énoncé de boucle car la répétition
#    est créée par des appels récursifs successifs. Il n'y a pas
#    de circularité car chaque fois l'appel s'applique à un argument
#    de plus en plus petit jusqu'au cas de base.

# Appeler la fonction principale
if __name__ == "__main__":
   main( sys.argv[1:] )
```

```
djmaya2-iro:Mise à niveau major$ ./factorielle.py -n 10 -v
 10 * factorielle( 9 )
  9 * factorielle( 8 )
   8 * factorielle( 7 )
    7 * factorielle( 6 )
     6 * factorielle( 5 )
      5 * factorielle( 4 )
       4 * factorielle( 3 )
        3 * factorielle( 2 )
         2 * factorielle( 1 )
          1 * factorielle( 0 )
La factorielle de 10 est 3628800 calculée en 8.702278137207031e-05 secondes
```

Tracer une fonction récursive

Trace de récursivité

- Une boîte pour chaque appel récursif
- Une flèche de chaque appelant à appelé
- Une flèche de chaque appelé à appelant montrant la valeur retournée



Analyze asymptotique d'un algorithme récursif

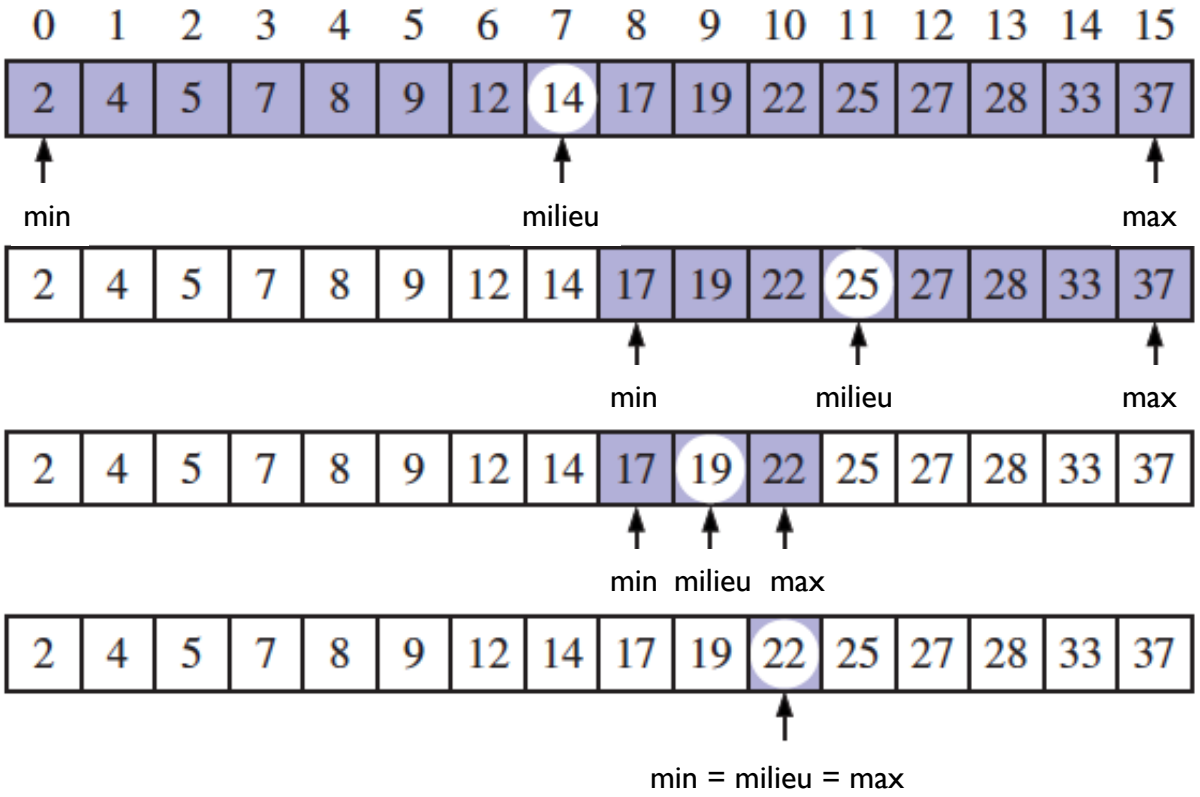
Avec un algorithme récursif, on compte chaque opération effectuée en fonction de l'activation particulière de la fonction qui gère le flux de contrôle au moment de son exécution. Ceci se généralise à des algorithmes non récursifs qui appellent d'autres fonctions de leur propre contexte.

Pour la fonction factorielle, on utilise la trace de récursion montrant les activations récursives.

On a vu un total de n+1 activations, alors que l'argument descent de n au premier appel, à n-1 dans le second, et ainsi de suite jusqu'au cas de base 0,

$\Rightarrow$ `factorielle( n )` est dans $O(n)$

Visualiser la recherche binaire



On considère trois cas :

- Si la cible égale data[milieu], alors on a trouvé la cible !
- Si la cible < data[milieu], alors on appel sur la première moitié de la séquence.
- Si la cible > data[milieu], alors on appel sur la deuxième moitié de la séquence.

```python
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "8 janvier 2014"
#
# Programme Python pour IFT2015/Mise à niveau/Initiation à Python
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

# Ce programme prend en input une valeur entière
# et la recherche dans une séquence indexable
# telle une liste en python

# utilise sys pour saisir les arguments de la ligne de commande
import sys

# utilise optparse pour parser la ligne de commande
from optparse import OptionParser

# Usage: factorielle.py [options]
#
# Options:
#   -h, --help     show this help message and exit
#   -n N, --n=N    entier entre 0 et 100 à chercher dans la liste
#   -v, --verbose  trace les appels récursifs de la recherche binaire

# Fonction principale
def main( argv ):

    data = [2,4,5,7,8,9,12,14,17,19,22,25,27,28,33,37]
    # Imprimer en output les data
    print( data )

    # variables globales pour les options et arguments
    global opts
    global args

    # mettre les options et arguments
    parser = OptionParser()
    parser.add_option( "-n", "--n", dest = "n", default = 0,
                       help = "entier entre 0 et 100 à chercher dans la liste", metavar = "N" )
    parser.add_option( "-v", "--verbose",
                       action = "store_true", dest = "verbose", default = False,
                       help = "trace les appels récursifs de la recherche binaire" )

    # parse the options and arguments
    opts, args = parser.parse_args()

    # s'assurer que n est un entien entre 0 et 100
    try:
        n = int( opts.n )
    except ValueError:
        print( opts.n, "n'est pas un entier !" )
        exit()

    # ici n est un entier
    # on s'assure qu'il est entre 0 et 100
    if n < 0 or n > 100:
        print( n, "doit être entre 0 et 100 !" )
        exit()
    else:
        # on cherche l'entier entré
        trouve = recherche_binaire( data, n, 0, len( data ) - 1, opts.verbose )
        if trouve is not None:
            print( "J'ai trouvé", n, "dans data à l'index", trouve, '!' )
        else:
            print( "Je n'ai pas trouvé", n, 'dans data !' )
```

```
# Fonction recherche binaire d'un élément cible dans une
# séquence de données implantée avec une liste. La liste,
# la cible, et les indices min et max qui bornent la recherche
# dans la séquence sont passés en arguments.
def recherche_binaire( data, cible, min, max, trace = False, profondeur = 0 ):
    if trace:
        print( profondeur * ' ', 'recherche_binaire(', data[min:max+1], ',', cible, ',', min, ',', max, ',', trace, ')', )
    if min > max:
        # liste vide
        return None #interval vide, pas de match
    else:
        # on essaye au milieu
        milieu = (min + max) // 2
        if cible == data[milieu]:
            # la cible est au milieu, eureka !
            return milieu
        elif cible < data[milieu]:
            # cible plus petite que la valeur au milieu
            # on cherche la portion gauche de la liste
            return recherche_binaire( data, cible, min, milieu-1, trace, profondeur+1 )
        else:
            # cible plus grande que la valeur au milieu
            # on cherche la portion droite de la liste
            return recherche_binaire( data, cible, milieu+1, max, trace, profondeur+1 )

# Appeler la fonction principale
if __name__ == "__main__":
   main( sys.argv[1:] )
```

```
djmaya2-iro:Mise à niveau major$ ./recherche_binaire.py -n 22 -v
[2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]
 recherche_binaire( [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37] , 22 , 0 , 15 , True )
  recherche_binaire( [17, 19, 22, 25, 27, 28, 33, 37] , 22 , 8 , 15 , True )
   recherche_binaire( [17, 19, 22] , 22 , 8 , 10 , True )
    recherche_binaire( [22] , 22 , 10 , 10 , True )
J'ai trouvé 22 dans data à l'index 10 !
```

La recherche binaire exécute en temps dans O(log n)

(pour une séquence triée de n éléments)

Pour prouver cela, on considère le fait qu'à chaque appel récursif le nombre d'éléments à parcourir est donné par

max - min + 1

Le nombre restant d'éléments est réduit par au moins la moitié à chaque appel récursif. Plus spécifiquement, de par la définition de **milieu**, le nombre d'éléments restants est soit :

$$(milieu - 1) - min + 1 = \lfloor (min+max)/2 \rfloor - min <= (max-min+1)/2$$

ou

$$max - (milieu + 1) + 1 = max - \lfloor (min+max)/2 \rfloor <= (max-min+1)/2$$

Initialement, le nombre d'éléments est n ; après le premier appel, il est au plus n/2 ; après le deuxième appel au plus n/4 ; et ainsi de suite.

Après le jème appel, le nombre d'éléments restants est au plus $n/2^j$. Dans le pire cas (celui d'une recherche infructueuse), les appels récursifs arrêtent quand il n'y a plus d'éléments.

Donc, le nombre maximum d'appels récursifs est le plus petit entier, r, tel que :

$n/2^r < 1$
$n < 2^r$
$\log n < r$; on prend le log de chaque côté

On peut donc prendre :

$r = \lfloor \log n \rfloor + 1,$

implicant que la recherche binaire exécute en temps dans O(log n). ■

Récursion de queue

Une récursion de queue est une fonction récursive où la fonction s'appelle elle-même à la fin ("queue") de la fonction dans laquelle aucun calcul n'est fait après le retour de l'appel récursif. Dans ce cas, plusieurs compilateurs remplacer cette récursion en itération.

La recherche binaire est un exemple de récursion de queue. Tous les appels récursifs sont à la fin de la fonction, c'est-à-dire qu'aucune instruction n'est exécutée au retour des appels récursifs.

Pour réimplémenter une récursion de queue par une itération, on enferme le corps dans une boucle et remplace les appels récursifs par des assignations de valeurs correspondantes aux substitutions d'arguments dans les appels récursifs.

```python
def recherche_binaire_iterative( data, cible ):
    min = 0
    max = len( data ) - 1
    while min <= max:
        milieu = ( min + max ) // 2
        if cible == data[ milieu ]:
            return True
        elif cible < data[ milieu ]:
            max = milieu - 1
        else:
            min = milieu + 1
    return False
```

```python
if min > max:
    return None #interval vide, pas de match
else:
    milieu = (min + max) // 2
    if cible == data[milieu]:
        return milieu
    elif cible < data[milieu]:
        #on cherche dans la portion gauche de la liste
        return recherche_binaire( data, cible, min, milieu-1, trace, profondeur+1 )
    else:
        #on cherche dans la portion droite de la liste
        return recherche_binaire( data, cible, milieu+1, max, trace, profondeur+1 )
```

```
djmaya2-iro:Mise à niveau major$ ./recherche_binaire_i.py -c 10000000
[2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]
10000000 itérations pour la version récursive a pris 14.05071210861206 secondes
10000000 itérations pour la version itérative a pris 8.139711141586304 secondes
```