Search&Sort 🔍

Présentation 📋

Ce projet, fait en python, permet de faire un tas de choses dans un tableau de réels parmi lesquelles:

- Une recherche dichotomique, ceci dans la mésure où le tableau est trié,
- Le tri du tableau (ordre croissant ou décroissant) via:
  - le tri rapide ou
  - le tri par tas

Structure 🏗️

La structure du projet se décline comme suit:

- `main.p`: c'est le fichier principal, celui qui une fois exécuté, interagit avec l'utilisateur de façon conviviale.

- `search.py`: c'est un module qui contient la fonction de recherche dichotomique.
- **`heap.py`**: c'est un module qui contient une classe Heap, permettant de voir un tableau comme un tas (arbre binaire) et ainsi de lui associer des paramètres et des méthodes utiles, parmi lesquelles la fonction de tri proprement dite, celle qui retiendra notra attention.
- **`sort.py`**: ce module contient l'implémentation du tri rapide avec la fonction de partitionnement, nécessaire à son fonctionnement.
- **`utils.py`**: ce dernier module, contient un ensemble de fonctions utilitaires, qui permmettent d'alléger le code principal, et d'offir une meilleure expérience à l'utilisateur.

Fonctionnement ⚙️

Comme expliqué plus haut le point d'netré du projet (fichier a exécuté est main.py, qui fait juste appels au outils définit dans les différents modules présentés plus haut, en vue de répondre aux besoins de recherche et de tri de l'utilisateur.

Explorons plus en détails le fonctionnement des différents modules:

N'hésitez pas d'utiliser la fontion help(...) de python pour plus d'infos sur un des modules présentés dans la suite

search.py

il contient une implémentation de l'algorithme de recherhce dichotomique dans un tableau déjà trié.

Elle est prototypée comme suit:

```
def dich_search(target: real, sequence: list[real], start_index: int = 0,
end_index: int = None) -> int:
    """
    Perform a binary search to find the index of 'target' in
'sequence'[start_index: end_index-1].

    Parameters:
    target (real): The value to search for within the sequence.
    sequence (list[real]): An indexable, ordered iterable to search within.
    start_index (int): The starting index of the search range.
    end_index (int): The ending index of the search range (exclusive).
```

```
    Returns:
    int: The index of 'target' in 'sequence' if found, otherwise -1.

    Time Complexity: O(log n), where n is the number of elements in the
search range.
    Space Complexity: O(1), as the space used does not depend on the size of
the input sequence.
    """
```

Elle utilise le principe de dichotomie pour rechercher l'élément désiré dans le tableau trié en le divisant successivement en deux moitié de taille égale.

Comme on peut voir son prototypage, elle se fait en:

- O(log n) pour la compléxité temporelleet en
- O(1) pour celle spatiale

sort.py

Ce module contient les fonctions nécessaires à l'implémentation du tri rapide parmi lesquelles

partition_list(...)

Cette dernière est prototypée comme suit:

```
def partition_list(array: list[real], start_index: int = 0, end_index: int =
None, order: int = 0, pivot_index: int = None) -> int:
"""
    Rearranges elements in a sublist of 'array' such that all elements less
than the pivot are before it,
    and all elements greater than or equal to the pivot are after it if
'order' is 0. If 'order' is 1, the
    opposite arrangement is applied. The function operates in-place and
returns the final index of the pivot element.

    Parameters:
    array (list[real]): The list of elements to be partitioned.
    start_index (int): The starting index of the sublist to partition.
    end_index (int): The ending index of the sublist to partition.
    order (int): Determines the order of arrangement; 0 for less than pivot
first, 1 for greater first.
    pivot_index (int): The index of the pivot element.

    Returns:
    int: The final index position of the pivot element after partitioning.
```

```
    Time Complexity: O(n), where n is the number of elements in the sublist.
    Space Complexity: O(1), as the rearrangement is done in-place without
using extra space.

    Note:
    The function optimizes space complexity by modifying the list in-place.
However, time complexity may
    worsen due to swaps.
    """
```

Elle permet suivant de placer le pivot à sa place dans le tableau suivant les valuers de `order`.

- Si `order` = 0, uniquement les éléments plus petits que pivaut sont avant lui
- Sinon, uniquement les éléments plus grands que pivaux seront avant lui

Cette fonction suivant les cas permettra ainsi d'implémenter à proprement dit le tri rapide pour trier un tableau donné dans le sens croissant ou décroissant.

Au vu de sa docstring, elle a une complexité:

- O(n) en terme de temps
- O(1) en terme d'espace vu que tout se passe sur place

quick_sort(...)

Elle est prototypée comme suit:

```
def quick_sort(datas: list[real], start_index: int = 0, end_index: int =
None, order: int = 0, pivot_index: int = None) -> None:
    """
    Sorts a sublist of 'datas' from 'start_index' to 'end_index' using the
Quick Sort algorithm.
    The sorting order is ascending if 'order' is 0 and descending if 'order'
is 1.

    Parameters:
    datas (list[real]): The list of elements to be sorted, which can contain
real numbers or strings.
    start_index (int): The starting index of the sublist to sort.
    end_index (int): The ending index of the sublist to sort; defaults to
the length of the list.
    order (int): Sorting order flag; 0 for ascending, 1 for descending.
    pivot_index (int): The index of the pivot element; defaults to the last
element of the sublist.

    Returns:
    None: The function sorts the list in place and does not return anything.
```

```
    Time Complexity: O(n log n) on average, where n is the number of
elements in the sublist.
    Space Complexity: O(log n), which is the stack space used by recursive
calls.

    Note:
    This implementation uses the 'partition_list' function to optimize space
complexity.
    """
```

Elle utilise le principe **DPR (Diviser Pour Régner)**, en partitionnant le tableau avec `partition_list(...)` vu plus haut, et en traint ainsi les sous tableaux récursivement. Le tri se fait suivant les valeurs de order comme expliqué plus haut

Comme on peut le constater dans sa docstring, ce tri a une complexité de:

- O(n log n) en terme de temps
- O(log n)en terme d'espace, vu l'occupation de la stack par les appels récursifs

heap.py

Ce module, de part son nom contient une classe Heap permettant de voir un tableau comme un tas et ainsi d'y effectuer un nombre d'opérations intéressantes, aboutissant à son tri par la méthode du tri par tas.

Cette classe est prototypée comme suit:

```
CLASSES
    builtins.object
        Heap

    class Heap(builtins.object)
     |  Heap(elements: list[int | float])
     |
     |  A Heap is a specialized tree-based data structure that satisfies the
heap property. For a max heap,
     |  this property ensures that for any given node I, the value of I is
greater than or equal to the values
     |  of its children. This implementation provides methods to build a
heap, maintain the heap property,
     |  sort the elements using the heap sort algorithm, and visually
represent the heap structure.
     |
     |  Attributes:
     |  elements (list[float]): The list of floating-point numbers that the
heap is built from.
     |  heap_size (int): The number of elements in the heap that need to be
```

```
maintained.
     |  total_elements (int): The total number of elements in the heap,
including those not currently in the heap structure.
     |
     |  Methods:
     |  build_heap(): Converts the list of elements into a heap.
     |  heapify(index: int): Ensures the subtree rooted at 'index' satisfies
the heap property.
     |  calculate_height(): Calculates the height of the heap.
     |  get_left_child(index: int): Gets the index of the left child of the
given node.
     |  get_parent(index: int): Gets the index of the parent of the given
node.
     |  get_right_child(index: int): Gets the index of the right child of
the given node.
     |  sort(): Sorts the elements in the heap using the heap sort
algorithm.
     |  print_elements(): Returns a string representation of the heap
elements.
     |  __str__(spacing: int, arrows: str): Generates a string
representation of the heap in a tree-like structure.
     |
     |  Time Complexity:
     |  - Building the heap: O(n log n)
     |  - Heapify operation: O(log n)
     |  - Calculating heap height: O(1)
     |  - Finding a child/parent index: O(1)
     |  - Heap sort: O(n log n)
     |  - String representation: O(n)
     |
     |  Space Complexity:
     |  - All operations: O(1) (in-place with no additional space required
except the input list)
     |  - String representation: O(n)
     |
     |  Example:
     |  >>> heap = Heap([3, 2, 1, 7, 8, 4, 10, 16, 12])
     |  >>> heap.sort()
     |  >>> print(heap.print_elements())
     |  [1, 2, 3, 4, 7, 8, 10, 12, 16]
     |  >>> print(heap)
     |                     1
     |                  ↙     ↘
     |              2              3
     |           ↙    ↘         ↙   ↘
     |         4       7       8       10
     |        ↙ ↘     ↙ ↘     ↙ ↘     ↙ ↘
     |      12  16
     |
     |  Methods defined here:
     |
     |  __init__(self, elements: list[int | float])
     |      Initializes a new Heap object.
     |
     |      Parameters:
     |      elements (list[real]): The list of floating-point numbers to be
turned into a heap.
```

```
     |
     |      Time Complexity: O(1), as it performs a constant number of
operations.
     |      Space Complexity: O(n), where n is the number of elements in the
input list.
     |
     |  __str__(self, spacing: int = 1, arrows: str = '↙↘') -> str
     |      Generates a string representation of the heap in a tree-like
structure.
     |
     |      Parameters:
     |      spacing (int): The number of spaces between elements in the
printed heap.
     |      arrows (str): The characters used to represent the tree
branches.
     |
     |      Returns:
     |      str: A string representation of the heap.
     |
     |      Time Complexity: O(n), where n is the number of elements in the
heap.
     |      Space Complexity: O(n), as it creates a string representation of
the heap.
     |
     |      Note:
     |      The method calculates the necessary spaces and arranges the
elements to visually represent
     |      the heap's tree structure. The width of elements and spacing are
adjusted to create a balanced look.
     |
     |  build_heap(self, order: int = 0) -> None
     |      Converts the list of elements into a max heap if order == 0, or
a mini heap else.
     |
     |      Time Complexity: O(n log n), where n is the number of elements
in the heap, in the worse case
     |      Space Complexity: O(1), as it modifies the list in place.
     |
     |  calculate_height(self) -> int
     |      Calculates the height of the heap.
     |
     |      Time Complexity: O(1), as it performs a constant number of
operations.
     |      Space Complexity: O(1), as it does not allocate any additional
space.
     |
     |      Returns:
     |      int: The height of the heap.
     |
     |  get_left_child(self, index: int) -> int
     |      Gets the index of the left child of the given node.
     |      Parameters:
     |      index (int): The index of the parent node.
     |
     |      Returns:
     |      int: The index of the left child.
     |
```

```
     |  get_parent(self, index: int) -> int
     |      Gets the index of the parent of the given node.
     |
     |      Parameters:
     |      index (int): The index of the child node.
     |
     |      Returns:
     |      int: The index of the parent.
     |
     |  get_right_child(self, index: int) -> int
     |      Gets the index of the right child of the given node.
     |
     |      Parameters:
     |      index (int): The index of the parent node.
     |
     |      Returns:
     |      int: The index of the right child.
     |
     |  heapify(self, index: int, order: int = 0) -> None
     |      Ensures the subtree rooted at 'index' satisfies the heap
property: a max heap if order == 0, or a min heap else.
     |
     |      Parameters:
     |      index (int): The root index of the subtree to heapify.
     |
     |      Time Complexity: O(log n), where n is the number of elements in
the heap.
     |      Space Complexity: O(log n), due to the recursive call stack.
     |
     |  print_elements(self) -> str
     |      Returns a string representation of the heap elements.
     |      Time Complexity: O(n), where n is the number of elements in the
heap.
     |      Space Complexity: O(n), as it creates a string representation of
the list.
     |
     |      Returns:
     |      str: The string representation of the heap elements.
     |
     |  sort(self, order: int = 0) -> None
     |      Sorts the elements in the heap using the heap sort algorithm.
     |      It sorts in the order of increasing if order == 0, or decreasing
if order == 1.
     |
     |      Time Complexity: O(n log n), where n is the number of elements
in the heap.
     |      Space Complexity: O(1), as it modifies the list in place.
     |
     |  ----------------------------------------------------------------------
--
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
```

```
DATA
    real = int | float
```

On peu constater qu'elle est assez bien documentée et illustrer, pour faciliter sa compréhension

Aussi comme la fontion `quick_sort(...)` plus haut, le tri se fait dans l'ordre croissant où décroissant suivant un paramètre `order.`

Ici, le tri se fait avec la méthode sort() et on a même la possibilité de visualiser graphiquement le tas à tout étape de son tri via la méthodé **str**() et donc en faisant juste un print(heap) pour une heap = Heap(...) donné, on a une représentation sympathique de sa structure en arbre semblable à celle ici en dessous:

```
>>> heap = Heap([3, 2, 1, 7, 8, 4, 10, 16, 12])
>>> heap.sort()
>>> print(heap.print_elements())
[1, 2, 3, 4, 7, 8, 10, 12, 16]
>>> print(heap)
                  1
               ↙     ↘
          2               3
        ↙   ↘           ↙   ↘
      4       7       8       10
     ↙ ↘     ↙ ↘     ↙ ↘     ↙ ↘
    12  16
```

utils.py

C'est juste un module qui contient des objets et méthodes utilitaires.

Il est assez simple et est prototypé comme suit:

```
FUNCTIONS
    bye() -> None
        Displays a farewell message and exits the program.

        Time Complexity: O(1), as it performs a constant number of
operations.
        Space Complexity: O(1), as it does not allocate any additional
space.

    control_exit(value: str) -> None
        Checks if the provided value is an empty string and triggers program
```

```
exit.

        Parameters:
        value (str): The string to check for emptiness.

        Time Complexity: O(1), as it checks a single condition.
        Space Complexity: O(1), as it uses no extra space.

        Returns:
        None: This function does not return a value; it exits the program if
the condition is met.

DATA
    real = int | float
    stop_message = 'Une saisie vide interrompt le programme'
```

Tests 🧪

SImulons différentes entrées de l'utilisateur et visualisons ensemble les résulatats:

Recherche Dichotomique

Cas d'une recherche fructueuse

```
*** Bienvenue dans Search&Sort ***

Objectif :
    - Ce programme offre la possibilité de faire une recherche dans un
tableau de réels trié avec l'algorithme de recherhce par dichotomie.    - Ce
dernier offre également la possibilité de trier un tableau de réels avec les
tris rapide et par tas.

Entrez l'opération à éffectuer :
    - 1: Recherche dichotomique
    - 2: Tri rapide
    - 3: Tri par tas
Réponse (Une saisie vide interrompt le programme): 1

!Le tableau à entrer doit être trié, sinon vous pourrez avoir des résultats
érronés!

Entrez les éléments du tableau sous la forme t1,t2,...,tn (sans espace ; la
virgule des réels est le '.' ; Une saisie vide interrompt le programme) :
1,7,8,11,17.8,84,744

Entrez l'élément à rechercher dans le tableau (Une saisie vide interrompt le
programme): 84
```

```
L'élément '84.0' est bien pas dans le tableau donné, à la position '6'.

Ravi de vous avoir servi! Pressez 'Entrer' pour quitter.
```

Cas d'une recherche infructueuse

```
*** Bienvenue dans Search&Sort ***

Objectif :
    - Ce programme offre la possibilité de faire une recherche dans un
tableau de réels trié avec l'algorithme de recherhce par dichotomie.    - Ce
dernier offre également la possibilité de trier un tableau de réels avec les
tris rapide et par tas.

Entrez l'opération à éffectuer :
    - 1: Recherche dichotomique
    - 2: Tri rapide
    - 3: Tri par tas
Réponse (Une saisie vide interrompt le programme): 1

!Le tableau à entrer doit être trié, sinon vous pourrez avoir des résultats
érronés!

Entrez les éléments du tableau sous la forme t1,t2,...,tn (sans espace ; la
virgule des réels est le '.' ; Une saisie vide interrompt le programme) :
1,7,8,11,17.8,84,744

Entrez l'élément à rechercher dans le tableau (Une saisie vide interrompt le
programme): 17.9

L'élément '17.9' n'est pas dans le tableau donné.

Ravi de vous avoir servi! Pressez 'Entrer' pour quitter.
```

Tri rapide

Tri rapide croissant

```
*** Bienvenue dans Search&Sort ***

Objectif :
    - Ce programme offre la possibilité de faire une recherche dans un
tableau de réels trié avec l'algorithme de recherhce par dichotomie.    - Ce
dernier offre également la possibilité de trier un tableau de réels avec les
tris rapide et par tas.

Entrez l'opération à éffectuer :
    - 1: Recherche dichotomique
    - 2: Tri rapide
```

```
    - 3: Tri par tas
Réponse (Une saisie vide interrompt le programme): 2

Entrez l'ordre du tri ('c' pour croissant et 'd' pour décroissant) ; Une
saisie vide interrompt le programme) : c

Entrez les éléments du tableau sous la forme t1,t2,...,tn (sans espace ; la
virgule des réels est le '.' ; Une saisie vide interrompt le programme) :
1,-7,0,2,0,-8,21,80,102,-8,10

Le tableau trié avec le tri rapide est: [-8.0, -8.0, -7.0, 0.0, 0.0, 1.0,
2.0, 10.0, 21.0, 80.0, 102.0].

Ravi de vous avoir servi! Pressez 'Entrer' pour quitter.
```

Tri rapide décroissant

```
*** Bienvenue dans Search&Sort ***

Objectif :
    - Ce programme offre la possibilité de faire une recherche dans un
tableau de réels trié avec l'algorithme de recherhce par dichotomie.    - Ce
dernier offre également la possibilité de trier un tableau de réels avec les
tris rapide et par tas.

Entrez l'opération à éffectuer :
    - 1: Recherche dichotomique
    - 2: Tri rapide
    - 3: Tri par tas
Réponse (Une saisie vide interrompt le programme): 2

Entrez l'ordre du tri ('c' pour croissant et 'd' pour décroissant) ; Une
saisie vide interrompt le programme) : d

Entrez les éléments du tableau sous la forme t1,t2,...,tn (sans espace ; la
virgule des réels est le '.' ; Une saisie vide interrompt le programme) :
1,-7,0,2,0,-8,21,80,102,-8,10

Le tableau trié avec le tri rapide est: [102.0, 80.0, 21.0, 10.0, 2.0, 1.0,
0.0, 0.0, -7.0, -8.0, -8.0].

Ravi de vous avoir servi! Pressez 'Entrer' pour quitter.
```

Tri par tas

Tri par tas croissant

```
*** Bienvenue dans Search&Sort ***
```

```
Objectif :
    - Ce programme offre la possibilité de faire une recherche dans un
tableau de réels trié avec l'algorithme de recherhce par dichotomie.    - Ce
dernier offre également la possibilité de trier un tableau de réels avec les
tris rapide et par tas.

Entrez l'opération à éffectuer :
    - 1: Recherche dichotomique
    - 2: Tri rapide
    - 3: Tri par tas
Réponse (Une saisie vide interrompt le programme): 3

Entrez l'ordre du tri ('c' pour croissant et 'd' pour décroissant) ; Une
saisie vide interrompt le programme) : c

Entrez les éléments du tableau sous la forme t1,t2,...,tn (sans espace ; la
virgule des réels est le '.' ; Une saisie vide interrompt le programme) :
1,-7,0,2,0,-8,21,80,102,-8,10

Le tableau trié avec le tris par tas est: [-8.0, -8.0, -7.0, 0.0, 0.0, 1.0,
2.0, 10.0, 21.0, 80.0, 102.0]

Le tableau trié sous forme de tas se présente comme suit:

                    -8.0
                ↙           ↘
          -8.0                    -7.0
        ↙     ↘                 ↙     ↘
     0.0         0.0         1.0         2.0
    ↙  ↘         ↙  ↘         ↙  ↘         ↙  ↘
 10.0  21.0  80.0 102.0

Ravi de vous avoir servi! Pressez 'Entrer' pour quitter.
```

Tri par tas décroissant

```
*** Bienvenue dans Search&Sort ***

Objectif :
    - Ce programme offre la possibilité de faire une recherche dans un
tableau de réels trié avec l'algorithme de recherhce par dichotomie.    - Ce
dernier offre également la possibilité de trier un tableau de réels avec les
tris rapide et par tas.

Entrez l'opération à éffectuer :
    - 1: Recherche dichotomique
    - 2: Tri rapide
    - 3: Tri par tas
Réponse (Une saisie vide interrompt le programme): 3

Entrez l'ordre du tri ('c' pour croissant et 'd' pour décroissant) ; Une
saisie vide interrompt le programme) : d

Entrez les éléments du tableau sous la forme t1,t2,...,tn (sans espace ; la
```

```
virgule des réels est le '.' ; Une saisie vide interrompt le programme) :
1,-7,0,2,0,-8,21,80,102,-8,10

Le tableau trié avec le tris par tas est: [102.0, 80.0, 21.0, 10.0, 2.0,
1.0, 0.0, 0.0, -7.0, -8.0, -8.0]

Le tableau trié sous forme de tas se présente comme suit:

                    102.0
                ↙           ↘
          80.0                    21.0
        ↙     ↘                ↙     ↘
     10.0        2.0         1.0         0.0
    ↙  ↘         ↙  ↘        ↙  ↘        ↙  ↘
 0.0   -7.0  -8.0  -8.0

Ravi de vous avoir servi! Pressez 'Entrer' pour quitter.
```

Autheur 👤

Ce projet a été réalisé par :

- Nom: KAMDEM POUOKAM
- Prénom: Ivann Harold
- Profession: Étudiant en 3GI-ENSPY
- E-mail: kapoivha@gmail.com

License 📜