Étude de la complexité des algorithmes de tri
Projet à rendre avant le 9 octobre 2024

1 Sujet

Le but du projet est de comparer la complexité des algorithmes de tri d'un tableau lorsqu'on utilise les algorithmes suivants :

- **Bubble sort**,
- **Selection sort**,
- **Insertion sort**,
- **Heap sort**,
- **Quick Sort**,
- **Merge sort**.

Ces algorithmes sont présentés en détail (en anglais) dans la partie 2.

Tableaux : La complexité des algorithmes de tri peut être évalué sur des tableaux :

- Triés par ordre croissant (donc déjà triés),
- Triés par ordre décroissant,
- remplis aléatoirement.

Selon le type d'algorithme de tri utilisé, le nombre d'opération peut être très différent pour ces différents types de tableaux.

Matériel fourni : Vous trouverez sur moodle un notebook python qui vous donnera une partie des codes nécessaires à votre projet. En particulier, le code fourni vous permet de

- générer les tableaux,
- mesurer et tracer le temps mis par un algorithme de tri.

Ce code est très basique et n'est là que pour vous aider à démarrer.

Travail à faire : En vous inspirant du notebook fourni et des éléments donnés en annexe, vous devrez

1. Implémenter les algorithmes (il y a pléthore de code à glaner sur internet ou avec chatgpt, cette partie est facile à faire),
2. Donner la complexité **théorique** de ces algorithmes et la vérifier **empiriquement**,
3. Identifier pour chaque algorithme le meilleur cas, le pire cas et le cas moyen à l'aide de simulations,
4. Comparer les algorithmes entre eux,
5. Expliquer de manière convaincante les résultats,
6. En déduire quelques règles de bonnes pratiques.

Rendu : Le rendu doit se faire avec **un seul** notebook qui contiendra le code et le texte dans des cellules markdown. Si vous développez beaucoup de code (création de classes, fonctions de tri, fonctions utilitaires, fonctions graphiques, etc.) vous pouvez alléger le notebook en en mettant une partie dans un fichier python *.py que vous importerez depuis le notebook.

Le nom du ficher à rendre devra être de la forme

```
projet_NOM1_Prénom1[_NOM2_Prenom2].ipynb
[projet_NOM1_Prénom1[_NOM2_Prenom2].py]
```

Entre [], les parties optionnelles.

Evaluation :

- La partie code seule ne rapporte **rien** !
- Il sera tenu compte de la qualité des commentaires et des illustrations.
- Faire une brève présentation de la théorie de la complexité.
- Pour tous les algorithmes, il est possible d'obtenir la complexité du pire cas de manière théorique, vous illustrerez la partie "cas moyen" par simulation, sauf goût prononcé pour les probabilités.

— Votre rendu doit comporter une partie comparaison de tous les algorithmes selon différents critères (petits tableaux, grands tableaux, pire cas, meilleur cas - s'il existe -, cas moyen, etc.).

Ce projet est à réaliser **seul** ou en **binôme**.

2 Annex : Sorting Algorithms

The following is a short presentation of the sorting algrothms you have to implement. Most of the text can be found somewhere on internet.

Selection sort

The algorithm divides the input list into two parts : a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Algorithm 1 Selection sort : In pseudocode the algorithm can be expressed as (0-based array) :

```
1:  function SelectionSort(A)
2:      n := length(A)
3:      for  i=0 to n-1 include do
4:          jmin := i
5:          for j =i+1 to n-1 do
6:              if A[j]<A[jmin] then
7:                  jmin := j
8:              end if
9:          end for
10:         if  jmin != i then swap(A[i], A[jmin])
11:         end if
12:         i := i+1
13:     end for
14: end function
```

Insertion sort

Insertion sort iterates, consuming one input element each repetition, and grows a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Algorithm 2 Insertion sort : In pseudocode the algorithm can be expressed as (0-based array) :

```
1:  function InsertionSort(A)
2:      n := length(A)
3:      i :=1
4:      while  i < n do
5:          j := i
6:          while j > 0 And A[j-1]>A[j] do
7:              swap(A[j-1], A[j])
8:              j := j-1
9:          end while
10:         i := i+1
11:     end while
12: end function
```

Optimized insertion sort After expanding the swap operation in-place as $x := A[j]$; $A[j] := A[j-1]$; $A[j-1] := x$ (where x is a temporary variable), a slightly faster version can be produced that moves $A[i]$ to its position in one go and only performs one assignment in the inner loop body :

Algorithm 3 Optimized insertion sort : In pseudocode the algorithm can be expressed as (0-based array) :

```
1: function InsertionSort(A)
2:     n := length(A)
3:     x := A[n]
4:     i :=1
5:     while  i < n do
6:         j := i
7:         while j ≥ 0 And A[j]>x do
8:             A[j+1] := A[j])
9:             j := j-1
10:        end while
11:        A[j+1] := x
12:    end while
13: end function
```

Bubble sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed. These passes through the list are repeated until no swaps had to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a comparison sort, is named for the way the larger elements "bubble" up to the top of the list.

Algorithm 4 Bubble sort : In pseudocode the algorithm can be expressed as (0-based array) :

```
1: function BubbleSort(A)
2:     n := length(A)
3:     repeat
4:         swapped := false
5:         for i := 1 to n-1 inclusive do do
6:             if A[i-1] > A[i] then then                    ▷ this pair is out of order
7:                 swap(A[i-1], A[i])                       ▷ swap them and remember something changed
8:                 swapped := true
9:             end if
10:        end for
11:    until not swapped
12: end function
```

Optimizing bubble sort : The bubble sort algorithm can be optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last $n-1$ items when running for the n-th time

Algorithm 5 Optimized bubble sort : In pseudocode the algorithm can be expressed as (0-based array) :

```
1: function BubbleSort(A)
2:     n := length(A)
3:     repeat
4:         swapped := false
5:         for i := 1 to n-1 inclusive do do
6:             if A[i-1] > A[i] then then                    ▷ this pair is out of order
7:                 swap(A[i-1], A[i])                       ▷ swap them and remember something changed
8:                 swapped := true
9:             end if
10:        end for
11:        n := n-1
12:    until not swapped
13: end function
```

Optimizing again bubble sort : More generally, it can happen that more than one element is placed in their final position on a single pass. In particular, after every pass, all elements after the last swap are sorted, and do not need to be checked again. This allows us to skip over many elements, resulting in about a worst case 50%

improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the "swapped" variable :

Algorithm 6 Doubly optimized bubble sort : In pseudocode the algorithm can be expressed as (0-based array) :

```
1:  function BubbleSort(A)
2:      n := length(A)
3:      repeat
4:          newn := 0
5:          swapped := false
6:          for i := 1 to n-1 inclusive do do
7:              if A[i-1] > A[i] then then
8:                  swap(A[i-1], A[i])
9:                  newn := i
10:             end if
11:         end for
12:         n := newn
13:     until n ≤ 1
14: end function
```

Heap sort

The heapsort algorithm can be divided into two phases : heap construction, and heap extraction.

The heap is an implicit data structure which takes no space beyond the array of objects to be sorted ; the array is interpreted as a complete binary tree where each array element is a node and each node's parent and child links are defined by simple arithmetic on the array indexes. For a zero-based array, the root node is stored at index 0, and the nodes linked to node i are

$$\begin{aligned} \text{iLeftChild}(i) &= 2i+1 \\ \text{iRightChild}(i) &= 2i+2 \\ \text{iParent}(i) &= \left\lfloor \frac{i-1}{2} \right\rfloor \end{aligned}$$

where the floor function $\lfloor x \rfloor$ rounds down to the preceding integer.

Algorithm 7 Heapify an array (0-based array) :

```
1:  function heapify(A, n, i)
2:      largest := i                                  ▷ Initialize largest as root
3:      l := 2i + 1
4:      r := 2i + 2
5:
6:      if l < n and A[i] < A[l] then
7:          largest := l
8:      end if
9:
10:     if r < n and A[largest] < A[r] then
11:         largest := r
12:     end if
13:
14:     if largest != i then
15:         swap(A[i], A[largest])
16:     end if
17:     heapify(A, n, largest)                        ▷ Heapify the root
18: end function
```

Although it is convenient to think of the two phases separately, most implementations combine the two. Two variables (here, start and end) keep track of the bounds of the heap area. The portion of the array before start is unsorted, while the portion beginning at end is sorted. Heap construction decreases start until it is zero, after which heap extraction decreases end until it is 1 and the array is entirely sorted.

Algorithm 8 Sort an array using a heap (0-based array) :

```
1: function heapSort(A)
2:     n := len(A)
3:     for i = n // 2 - 1 to -1 by -1 do  ▷ Since last parent will be at ((n//2)-1) we can start at that location
4:         heapify(A, n, i)
5:     end for
6:
7:     for i = n - 1 to 0 by -1 do                                  ▷ One by one extract elements
8:         swap(A[i], A[0])
9:         heapify(A, i, 0)
10:    end forreturn A
11: end function
```

Quick sort

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Algorithm 9 Sort an array using QickSort :

```
1: function QuickSort(A, start, end)
2:     if start >= end then return
3:     end if
4:     iPivot = Partition(A, start, end)
5:     QuickSort(A, start, iPivot − 1)
6:     QuickSort(A, iPivot + 1, end)
7: end function
8:
9: function Partition(A, start, end)
10:    val = A[end]
11:    iPivot = start
12:    for i=start to end do
13:        if A[i] <= val then
14:            t = A[i]
15:            A[i] = A[iPivot]
16:            A[iPivot] = t
17:            iPivot = iPivot + 1
18:        end if
19:    end for
20:    return iPivot − 1
21: end function
```

Merge sort

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

There exists two implementation of the merge sort `top_down` and `bottom-up`.