recherche du minimum(ou du maximum)

Recherche_minimum dans un tableau à n éléments

Entrée : Un tableau A de n nombres $[a_1, .., a_n]$
Sortie : Indice du plus petit élément

```
n=taille(A)
min ← 1
Pour i allant de 2 à n faire :
      Si A[i] < A[min] :
         min ← i
       Fin Si
Fin Pour
Retourner min
```

Exercice:Modifier l'algorithme ci-dessus pour déterminer le plus grand élément dans le tableau A.

Complexité: Dans cet algorithme il y a obligatoirement $n-1$ comparaisons.
On a donc un coût dont l'ordre de grandeur est n (coût linéaire), la taille du tableau .On écrit $O(n)$.

Présentation d'un tri , le tri sélection , qui ordonne les éléments d'un tableau (liste en Python)

-

TRI SELECTION

Présentation d'un tri , le tri sélection , qui ordonne les éléments d'un tableau (liste en Python)

-
- Trouver le plus petit élément et le mettre au début de la liste.

Présentation d'un tri , le tri sélection , qui ordonne les éléments d'un tableau (liste en Python)

-
- Trouver le plus petit élément et le mettre au début de la liste.
- Trouver le deuxième plus petit élément et le mettre en deuxième position.

TRI SELECTION

Présentation d'un tri , le tri sélection , qui ordonne les éléments d'un tableau (liste en Python)

-
- Trouver le plus petit élément et le mettre au début de la liste.
- Trouver le deuxième plus petit élément et le mettre en deuxième position.
- Trouver le troisième plus petit élément et le mettre en troisième position.

Présentation d'un tri , le tri sélection , qui ordonne les éléments d'un tableau (liste en Python)

-
- Trouver le plus petit élément et le mettre au début de la liste.
- Trouver le deuxième plus petit élément et le mettre en deuxième position.
- Trouver le troisième plus petit élément et le mettre en troisième position.
- ...

8 12 21 14 3

TRI SELECTION: Un exemple

8	12	21	14	3
3	12	21	14	8

TRI SELECTION: Un exemple

8	12	21	14	3
3	12	21	14	8
3	8	21	14	12

TRI SELECTION: Un exemple

8	12	21	14	3
3	12	21	14	8
3	8	21	14	12
3	8	12	14	21

TRI SELECTION: Un exemple

8	12	21	14	3
3	12	21	14	8
3	8	21	14	12
3	8	12	14	21

TRI SELECTION

TRI_SELECTION(A)
Entrée : Un tableau de n nombres $(a_1, .., a_n)$
Sortie : une permutation de la suite de telle sorte que $a_1 \leqslant a_2 \leqslant ... \leqslant a_n$.

```
Pour i allant de 1 à n-1 faire
    min ← i
    Pour j allant de i+1 à n faire
        SI A[j] < A[min]
            min ← j
        Fin SI
    Fin Pour
    Echange(A,i,min)
Fin Pour
```

Remarque : l'appel de Echange(A,i,min) permute les éléments A[i] et A[min] du tableau.

Complexité du tri sélection

La boucle de la ligne 1 doit être exécutée n-1 fois. La boucle de la ligne 3 effectue $n-1$ comparaisons au premier passage dans la boucle , puis $n-2$, $n-3$,...,comparaisons , ce qui donne au total: $(n-1)+(n-2)+...+1$ comparaisons. D'après le cours sur les suites arithmétiques nous obtenons:

$\dfrac{n(n-1)}{2}$ comparaisons

On dit que l'algorithme de Tri par sélection a une complexité en n^2.On écrit $O(n^2)$.

Remarque: Pour déterminer la complexité de l'algorithme du tri sélection , on s'est intéressé ici aux seules comparaisons (ligne 4). On peut se demander pourquoi ne s'intéresse-t-on pas aux autres opérations (affectations aux lignes 2 ,5 ainsi que l'échange à la ligne 8).
En réalité , compter ces opérations ne change pas la complexité de l'algorithme qui sera toujours quadratique.

Proposer une explication.

Tri du joueur de cartes

- Ordonner les deux premiers éléments.

Tri du joueur de cartes

- Ordonner les deux premiers éléments.
- Insérer le troisième élément de sorte que les trois premiers éléments soient rangés dans le bon ordre.

Tri du joueur de cartes

- Ordonner les deux premiers éléments.
- Insérer le troisième élément de sorte que les trois premiers éléments soient rangés dans le bon ordre.
- Puis on insère le quatrième élément à sa place

Tri du joueur de cartes

- Ordonner les deux premiers éléments.
- Insérer le troisième élément de sorte que les trois premiers éléments soient rangés dans le bon ordre.
- Puis on insère le quatrième élément à sa place
- ...

A la fin de la ième insertion, les i premiers éléments de T sont triés et rangés au début du tableau.

TRI INSERTION

TRI_Insertion(A)
Entrée : Une tableau de n nombres $(a_1, .., a_n)$
Sortie : une permutation de la suite de telle sorte que $a_1 \leqslant a_2 \leqslant ... \leqslant a_n$.

```
n=taille(A)
Pour j allant de 2 à A.longueur faire :
    cle ← A[j]
   // Insere A[j] dans la suite triee A[1,..,j-1]
    i ←j-1
    Tant que i >0 et A[i]>cle Faire :
          A[i+1]← A[i]
          i← i-1
    A[i+1] ←cle
```

Exercice: Déterminer la complexité du tri insertion

Pour montrer que l'algorithme de tri par insertion est **correct** , on se sert de la notion **d'invariant de boucle**.

Invariant de boucle(Correction)

On appelle invariant de boucle , une propriété qui , si elle est vraie avant l'entrée dans la boucle reste vraie après chaque passage dans la boucle , et est par conséquent vraie aussi à la sortie de la boucle.

La mise en évidence d'un invariant de boucle adapté permet de prouver la **correction** d'un algorithme.

Pour montrer qu'un programme se termine , on se sert de la notion de Variant.

Variant de boucle (Terminaison)

il s'agit d'une quantité entière qui :

- Doit être positive ou nulle pour rester dans la boucle.
- Doit décroître strictement à chaque itération.

Si on arrive à trouver une telle quantité, il est évident que l'on va nécessairement sortir de la boucle au bout d'un nombre fini d'itérations, puisqu'un entier positif ne peut décroitre infiniment.

Pour l'algorithme de tri par insertion , on remarque que le sous-tableau A[1..j-1] possède les propriétés d'un invariant.
En effet , les éléments de ce sous-tableau possèdent la propriété d'être triés **au début** , **avant** chaque itération de la boucle for et **à la fin**.
Ainsi:

Lorsque j=2 , le sous tableau A[1..j-1] est le sous tableau A[1] qui contient le premier élément du tableau à trier et qui de fait est déjà trié.

TRI INSERTION-Terminaison-Correction

Montrons que chaque itération conserve l'invariant:

Montrons que chaque itération conserve l'invariant:

Dans la **boucle Pour** , A[j-1] , A[j-2] , A[j-3] etc , se déplacent d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour insérer A[j].

Montrons que chaque itération conserve l'invariant:

Dans la **boucle Pour** , A[j-1] , A[j-2] , A[j-3] etc , se déplacent d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour insérer A[j].

Le sous-tableau A[1..j] se compose alors des éléments situés initialement dans A[1..j], mais qui ont été triés.

Montrons que chaque itération conserve l'invariant:

Dans la **boucle Pour** , A[j-1] , A[j-2] , A[j-3] etc , se déplacent d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour insérer A[j].

Le sous-tableau A[1..j] se compose alors des éléments situés initialement dans A[1..j], mais qui ont été triés.

Intéressons-nous à présent à la boucle intérieure Tant que.

Pour montrer que la **boucle Tant que** termine , on peut remarquer que la condition i $>$ 0 est mise en défaut dès que i atteint la valeur 0 , ce qui est certain puisque l'instruction i=i-1 décrémente la valeur de i d'une unité (d'où ici la notion de variant) et comme le tableau contient un nombre fini d'éléments , on est sûr que i atteint la valeur 0.

Pour montrer que la **boucle Tant que** termine , on peut remarquer que la condition i $>$ 0 est mise en défaut dès que i atteint la valeur 0 , ce qui est certain puisque l'instruction i=i-1 décrémente la valeur de i d'une unité (d'où ici la notion de variant) et comme le tableau contient un nombre fini d'éléments , on est sûr que i atteint la valeur 0.

Quand la boucle Pour se termine , c'est-à-dire lorsque j=n+1 , alors le sous-tableau A[1..j-1] est le tableau entier A[1..n] et il est trié.