Algorithmes de tri

ITC PC

M. Charles

Exercices

Exercices

- Écrire une fonction Python qui affichent les n premiers carrés en partant de 1.

- Écrire une fonction Python de signature `suite(n: int) -> float` qui prend en paramètre un nombre n et qui renvoie u_n défini par :

$$u_0 = 2$$
$$u_{n+1} = 1 - e^{-u_n}$$

Exercices

- Écrire une fonction Python qui affichent les n premiers carrés en partant de 1.

```python
def carres(n: int):
    for i in range(1,n+1):
        print(i*i)
```

- Écrire une fonction Python de signature `suite(n: int) -> float` qui prend en paramètre un nombre n et qui renvoie u_n défini par :

$$u_0 = 2$$
$$u_{n+1} = 1 - e^{-u_n}$$

Exercices

- Écrire une fonction Python qui affichent les n premiers carrés en partant de 1.

```
def carres(n: int):
    for i in range(1,n+1):
        print(i*i)
```

- Écrire une fonction Python de signature `suite(n: int) -> float` qui prend en paramètre un nombre n et qui renvoie u_n défini par :

$$u_0 = 2$$
$$u_{n+1} = 1 - e^{-u_n}$$

```
import math
def suite(n: int) -> float:
    u = 2
    for i in range(n):
        u = 1 - math.exp(-u)
    return u
```

Tri par bulles

Détails

```
def balayage(L: list) -> None:
    # Une passe où on corrige les inversions rencontrées
    ...
def tri_bulles(L: list) -> None:
    for i in range(len(L)):
        balayage(L, len(L)-i)
```

Exercice : Compléter le code de `balayage`.

Détails

```
def balayage(L: list, last: int) -> None:
    for i in range(last-1):
        if L[i] > L[i+1]:
            L[i], L[i+1] = L[i+1], L[i]

def tri_bulles(L: list) -> None:
    for i in range(len(L)):
        balayage(L, len(L)-i)
```

Exercice : Compléter le code de `balayage`.

Analyse de complexité du balayage

```python
def balayage(L: list, last: int) -> None:
    for i in range(last - 1):
        if L[i] > L[i+1]:
            L[i], L[i+1] = L[i+1], L[i]
```

Liste de taille n.

- Comparaisons :
- Espace supplémentaire requis :

Analyse de complexité du balayage

```python
def balayage(L: list, last: int) -> None:
    for i in range(last - 1):
        if L[i] > L[i+1]:
            L[i], L[i+1] = L[i+1], L[i]
```

Liste de taille n.

- Comparaisons : $O(n)$
- Espace supplémentaire requis : $O(1)$

Analyse de complexité du tri par bulles

Liste de taille n. Balayage :

- comparaisons : $O(n)$
- espace supplémentaire requis : $O(1)$

```
def tri_bulles(L: list) -> None:
    for i in range(len(L)):
        balayage(L,len(L)-i)
```

Tri :

- comparaisons :
- espace supplémentaire requis :

Analyse de complexité du tri par bulles

Liste de taille n. Balayage :

- comparaisons : $O(n)$
- espace supplémentaire requis : $O(1)$

```
def tri_bulles(L: list) -> None:
    for i in range(len(L)):
        balayage(L,len(L)-i)
```

Tri :

- comparaisons : $O(n^2)$
- espace supplémentaire requis : $O(1)$

Analyse de complexité du tri par bulles

Liste de taille n. Balayage :

- comparaisons : $O(n)$
- espace supplémentaire requis : $O(1)$

```
def tri_bulles(L: list) -> None:
    for i in range(len(L)):
        balayage(L,len(L)-i)
```

Tri :

- comparaisons : $O(n^2)$
- espace supplémentaire requis : $O(1)$

Remarque : Tri destructif, en place. Se voit à la signature :

```
tri_bulles(L: list) -> None
```

Tri par sélection

Détails

```
def indice_du_min(L: list, a: int) -> int:
    # Renvoie une position après a où L[a:] est minimale
    ...
def tri_sélection(L: list) -> None:
  for i in range(len(L)):
      j = indice_du_min(L, i)
      L[i], L[j] = L[j], L[i]
```

Exercice : Compléter la fonction `indice_du_min`.

Détails

```
def indice_du_min(L: list, a: int) -> int:
    i = a
    for j in range(a, len(L)):
        if L[j] < L[i]:
            i = j
    return i

def tri_sélection(L: list) -> None:
  for i in range(len(L)):
      j = indice_du_min(L, i)
      L[i], L[j] = L[j], L[i]
```

Exercice : Compléter la fonction `indice_du_min`.

Analyse de complexité de la localisation du min

```python
def indice_du_min(L: list, a: int) -> int:
    i = a
    for j in range(a, len(L)):
        if L[j] < L[i]:
            i = j
    return i
```

- Comparaisons :
- Espace supplémentaire requis :

Analyse de complexité de la localisation du min

```python
def indice_du_min(L: list, a: int) -> int:
    i = a
    for j in range(a, len(L)):
        if L[j] < L[i]:
            i = j
    return i
```

- Comparaisons : $O(n)$
- Espace supplémentaire requis : $O(1)$

Analyse de complexité du tri par sélection

Indice du min :

- comparaisons : $O(n)$
- espace supplémentaire requis : $O(1)$

```
def tri_sélection(L: list) -> None:
    for i in range(len(L)):
        j = indice_du_min(L, i)
        L[i], L[j] = L[j], L[i]
```

Tri :

- comparaisons :
- espace supplémentaire requis :

Analyse de complexité du tri par sélection

Indice du min :

- comparaisons : $O(n)$
- espace supplémentaire requis : $O(1)$

```
def tri_sélection(L: list) -> None:
    for i in range(len(L)):
        j = indice_du_min(L, i)
        L[i], L[j] = L[j], L[i]
```

Tri :

- comparaisons : $O(n^2)$
- espace supplémentaire requis : $O(1)$

Analyse de complexité du tri par sélection

Indice du min :

- comparaisons : $O(n)$
- espace supplémentaire requis : $O(1)$

```
def tri_sélection(L: list) -> None:
    for i in range(len(L)):
        j = indice_du_min(L, i)
        L[i], L[j] = L[j], L[i]
```

Tri :

- comparaisons : $O(n^2)$
- espace supplémentaire requis : $O(1)$

Remarque : Tri destructif, en place. Se voit avec la signature :

```
tri_sélection(L: list) -> None
```

On peut envisager une version hors-place à base de `min`, `remove` et `append`, plus simple à écrire mais cachant davantage de complexité dans les fonctions natives :

```
def tri_sélection_hors_place(L: list) -> list:
    M = []
    while L:
        x = min(L)
        L.remove(x)     # tri destructif,
        M.append(x)     # hors-place
    return M
```

Tri fusion

Détails

```
def fusion(G: list, D: list) -> list:
    # fusionne les deux listes G et D,
    # supposées triées par ordre croissant
    ...

def tri_fusion(L: list) -> list:
    n = len(L)
    if n <= 1:
        return L
    else:
        m = n // 2
        G = tri_fusion(L[:m])
        D = tri_fusion(L[m:])
        return fusion(G, D)
```

Exercice : Compléter la fonction `fusion`

Détails

```
def fusion(G: list, D: list) -> list:
    # fusionne les deux listes G et D,
    # supposées triées par ordre croissant
    i, j = 0, 0
    R = []
    while i < len(G) and j < len(D):
        if G[i] < D[j]:
            R.append(G[i])
            i += 1
        else:
            R.append(D[j])
            j += 1
    return R + G[i:] + D[j:]
```

Analyse de complexité

- Fusion (n = taille du résultat)
  - comparaisons :
  - espace supplémentaire requis :
- Tri

Analyse de complexité

- Fusion ($n =$ taille du résultat)
  - comparaisons : $O(n)$
  - espace supplémentaire requis : $O(n)$
- Tri

$$C(n) \leqslant 2\,C\left(\frac{n}{2}\right) + O(n)$$

$$C(2^n) \leqslant \ldots$$

Analyse de complexité

- Fusion ($n =$ taille du résultat)
  - comparaisons :
  - espace supplémentaire requis :
- Tri

$$C(n) \leqslant 2\,C\left(\frac{n}{2}\right) + O(n)$$

$$C(2^n) \leqslant \ldots$$

Remarque : Tri conservatif, hors-place.

```
tri_fusion(L: list) -> list
```

Comparaison des algorithmes

Comparaison des algorithmes

Algorithme	#Comparaisons	Espace	Catégories
Bulles	$O(n^2)$	$O(1)$	itératif, en place
Sélection	$O(n^2)$	$O(1)$	itératif, en place
Fusion	$O(n \log n)$	$O(n \log n)$	récursif, hors-place