cours_5

March 10, 2021

Cours 5 (TRI fin)

Le cours du jour traite des points suivants:

- Tri rapide
- Partitionnement
- Choix du partionnement
- Conclusion Tri

0.1 Stratégie Diviser pour Régner

0.1.1 Tri rapide (quicksort)

- Utilisation du principe diviser pour régner
- Basé sur un partitionnement guidé des données (par rapport à imposé pour tri fusion)

[59]:
```
def trirapide(T, g=0, d=None):
    "Tri de T[g,d]"
    if d is None:
        d = len(T)-1
    if g < d:
        ipivot = partition(T, g, d)
        T = trirapide(T, g, ipivot-1)
        T = trirapide(T, ipivot+1, d)
    return T
```

Algorithme 6: Partition

Données : T un tableau, g, d deux indices du tableau
Résultat : la position i ($\in [g, d]$) telle que $T[j] \leq T[i]$ pour $g \leq j \leq i$ et $T[i] < T[j]$ pour $i < j \leq d$

pivot $\leftarrow$ T[d] ; i $\leftarrow$ g;
pour $j \leftarrow g$ *à* d*-1* **faire**
    si *T[j]* $\leq$ *T[d]* **alors**
        Echanger T[i] et T[j];
        i $\leftarrow$ i+1;
    fin
fin
Echanger T[i] et T[d];
return i;

Tri rapide: partitionnement

[60]:
```python
def partition(T, g, d):
    "Partition de T[g..d] en T[g..i] <= pivot et T[i+1..d] > pivot "
    pivot = T[d]
    i = g
    for j in range(g, d):
        if T[j] <= pivot:
            T[i], T[j] = T[j], T[i]
            i = i + 1
    T[i], T[d] = T[d], T[i]     # le pivot va en T[i]
    return i
```

Tri rapide: Invariant de partition Propriété:

Tous les éléments de $[d, i[$ sont $\leq$ à pivot Tous les éléments de $[i, j[$ sont $>$ à pivot

- Initialisation: i = j = g
  OK
- Récurrence:
  - Si T[j] <= pivot:
    - Echange de T[i] (> pivot) et T[j] ($\leq$ pivot)
    - i = i + 1, j = j + 1 => Proprieté préservée
  - Si T[j] > pivot:
    - j = j + 1 => Proprieté préservée
- En fin, échange de T[i] (> pivot) et T[d] = pivot => Proprieté préservée

Implémentation

[61]:
```python
### Vérification rapide

T = [3, 5, -1, 4, 7, 9, 4, 2]
```

```python
print(trirapide(T))
```

```
[-1, 2, 3, 4, 4, 5, 7, 9]
```

Tri rapide : Bon/Mauvais partitionnement

- Qu'est ce qu'un bon partitionnement pour le tri rapide ?

Equilibré : n/2 éléments à gauche et à droite.

- Qu'est ce qu'un mauvais partitionnement pour le tri rapide ?

Deséquilibré : n-1 et 1 élements

- Tout cela dépend:
  - la stratégie de choix du pivot.
  - du contenu tableau… humm pourquoi ?

Tri rapide : Choix du pivot

- Choix systématique: premier, dernier élément => mauvais
- Choix systématique: la valeur de l'élément au milieu => pourquoi pas ?

- La médiane de trois nombres tirés au hasard => bien mieux

Voir les codes.

Tri rapide : Analyse du tri

- Si le partitionnement est mauvais:

$$Ops(n) = Ops(n-1) + Ops(1) + n$$

$$... \Rightarrow Ops(n) \in O(n^2)$$

- Si le partitionnement est parfait:

$$Ops(n) = 2\ Ops(n/2) + n$$

$$... \Rightarrow Ops(n) \in \Omega(nlog_2(n))$$

- Analyse en moyenne (hors cadre du cours)

$$E[Ops(n)] \in \Theta(nlog_2(n))$$

(Bonus) Tri rapide : Problème

- Question: Existe t'il toujours une stratégie donnant lieu à un partitionnement équilibré ???

- Réponse: Non !!! si le tableau contient de nombreux doublons
- Conséquences: Explosion du nombre d'appels récursifs
- Solution: Améliorer le partitionnement

Découper en trois parties:

- les éléments <

- les éléments $=$
- les éléments $>$

et ne trier récursivement que les deux extrémités.

Exercice: Ecrire une telle fonction de partitionnement (*Connu également sous le nom de "Drapeau hollandais"*)

[4]:
```python
def partition3(T, g, d):
    "Drapeau Hollandais"
    pivot = T[d]
    i = j = g
    ### A compléter
    return i, j
```

[36]:
```python
def partition3(T, g, d):
    "Drapeau Hollandais (pas en place)"
    "Partition de T[g..d] en T[g..i[ < pivot, T[i..j[ = pivot,  T[j..d] > pivot"
    pivot = T[d]
    i = j = g
    for k in range(g, d):
        if T[k] < pivot:
            if i < j < k:
                T[i], T[j], T[k] = T[k], T[i], T[j]  # n'est plus en place
            elif i < j and j == k:
                T[i], T[k] = T[k], T[i]
            elif i == j and j < k:
                T[i], T[k] = T[k], T[i]
            i = i + 1
            j = j + 1
        elif T[k] == pivot:
            T[j], T[k] = T[k], T[j]
            j = j + 1
    T[j], T[d] = T[d], T[j]    # le pivot va en T[j]  # n'est plus en place
    return i, j+1
```

[62]:
```python
T = [1, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5, 2, -1]
pivot = T[-1]

Tinf = [Ti for Ti in T if Ti < pivot]
Teq = [Ti for Ti in T if Ti == pivot]
Tsup = [Ti for Ti in T if Ti > pivot]

print(Tinf, Teq, Tsup)
i, j = partition3(T, 0, len(T)-1)
print(f"{T[:i]}, {T[i:j]}, {T[j:]}")
```

```
[-4, -2] [-1, -1] [1, 2, 7, 2, 5, 9, 2, 13, 5, 2]
[-4, -2], [-1, -1], [2, 5, 9, 1, 2, 13, 2, 5, 2, 7]
```

Tri rapide : Conclusion

- Tri par comparaison, pas stable, en place, récursif
- Pire des cas: $O(n^2)$
- Meilleur des cas: Ω(n log n)
- Efficience proche de l'optimal (n log n) même en moyenne
- Il existe de bonnes stratégies:
  - de choix de pivot (Ex.: médiane de 3)
  - de partitionnement en 3 parties pour éviter le problème de doublons
- Amélioration: pour des tableaux de petite taille -> Tri par insertion

0.1.2 Conclusion sur les tris

Tri	meilleur	pire	Moyenne	Commentaires
par sélection	n^2	n^2	n^2	Inefficace
par insertion	n	n^2	n^2	Eff. petits tableaux
fusion	$n \log n$	n log n	$n \log n$	Mém. aux. + dépl.

(suite)

Tri	meilleur	pire	Moyenne	Commentaires
rapide	n log n	n^2	n log n	proche de l'opt., - dépl.
Timsort	n	n log n	n log n	meilleur mais mém. aux.
par comptage (TP)	n+k	n+k	n+k	Pas de comp., mém. aux.

0.2 Recherche du k-ième élément

Définition du problème:

Entrée: T un tableau de n éléments (distincts) et k un nombre tel que 1 <= k <= n

Sortie: La valeur x appartenant à T tel que k-1 éléments de T sont inférieurs à x

Exemples d'applications

- Quel est l'âge médian en France ? (40,8 contre 41,7 en moy.)
- Combien gagne les 10% les plus riches ?
- Détection les événements rares

0.2.1 Exemples triviaux: Recherche min ou max

- k = 1 -> min
- k = n -> max

[6]:
```
def minimum(T):
    x = T[0]
    for e in T[1:]:
        if e < x:
            x = e
```

```
    return x
```

Analyse: n-1 comparaisons au minimum => $\Theta(n)$, linéaire

0.2.2 Exemples triviaux: Recherche min *et* max

[40]:
```
def minmax(T):
    min, max = T[0], T[0]
    for e in T[1:]:
        if e < min:  min = e
        if e > max:  max = e
    return min, max
```

Analyse: 2*(n-1) comparaisons au minimum => $\Theta(n)$, linéaire

Recherche min *et* max : Faire mieux Supposons T[i] et T[j] deux éléments de T.

Dans l'algorithme précédent, 4 comparaisons pour calculer

- minimum(T[i], T[j], min) : 2 comp.
- maximum(T[i], T[j], max) : 2 comp.

Remarquons:

- minimum(T[i], T[j]) **et** maximum(T[i], T[j]): 1 comp.
- minimum(minimum(T[i], T[j]), min): 1 comp.
- maximum(maximum(T[i], T[j]), max): 1 comp.

Donc 3 comparaisons au lieu de 4 !

Stratégie

- Prendre deux éléments successifs T[i] et T[i+1]
- Mettre à jour le min et max en 3 comparaisons
- Gérer les cas limites:
  - Tableau à 1 élément
  - Nombre impair d'éléments

[8]:
```
def minmax_3(T):
    if len(T) == 0:
        return None, None
    i = 1 if len(T) % 2 == 1 else 0
    min =  max = T[0]
    while i + 1 <= len(T) - 1:
        if T[i] <= T[i+1]:    # T[i] < T[i+1]
            if T[i] < min:
                min = T[i]
            if T[i+1] > max:
                max = T[i+1]
        else:                 # T[i] >= T[i+1]
            if T[i+1] < min:
```

```
                min = T[i+1]
            if T[i] > max:
                max = T[i]
        i = i + 2
    return min, max
```

[9]:
```
T = [1, 5, -1, 2, 7, 2, 5, 9, -4, 2, 13, -2, 5, 2]
print(minmax_3(T))
```

```
(-4, 13)
```

Analyse:

- Pour tout couple: 3 comparaisons
- Soit env. 3*(n//2)=1.5*n comp. au lieu de 2*n, l'algorithme reste linéaire.

Attention:

- les fonctions natives min et max de Python sont implementées en C
- la fonction minmax n'existe pas dans le langage

Recherche min et max : approche récursive

- Découpage en 2 parties égales
- Calcul des min **et** max à gauche et à droite
- Fusion des résultats
- Simple, clair …
- Mais lent !!