Algo

- Numero: 6
- Prof: Alexandre Duret-Lutz
- Date: 26 Octobre 2017

Optimisation de QuickSort

Les optimisations dont on va parler ne vont rien changer a la complexite mais la vitesse d'execution sera meilleur.

De plus le quick sort affecte localement le tableau. Contrairement au heap sort qui lui va se balader un peu partout dans le tableau pour le trier. Hors le processeur prefere la manipulation de memoire contigue.

Optimisation du pivot

On veut trouver un meilleur pivot. L'ideal c'est la medianne. Le probleme c'est qu'elle n'est pas simple ni rapide a trouver.

Un truc qu'on peut faire c'est prendre `n` valeur reparti dans la tableau. Puis on prend la medianne de ces valeurs. Une valeur bien de `n` est `3`. Car la medianne est simple a calduler.

Ca permet de luter contre un mauvais choix de pivot. Mais elle retire pas le pire cas.

Cet exemple montre un pire cas lorsqu'on applique l'algo avec `n = 3`.

0	8	2	10	4	12	6	**1**	3	5	7	9	11	**13**

En pratique tout le monte implemente quand meme le quicksort avec ca.

Si on a besoin de securite pour un serveur par exemple. Ca a du sens de prendre un pivot aleatoire pour ne plus etre previsible. Le defaut est que `random` est une fonction assez lourd. De plus on creer `n - 1` partition dans `QuickSort` du coup on appelle beaucoup de fois `random`. On peut faire un `shuffle` avant de faire un quicksort ca revient au meme en terme de nombre d'appelle a `random`.

Conclusion: Il faut comme pivot la medianne de trois valeurs reparti dans le tableau.

```
1 def Partition(A, b, e):
2       p = medianne3(A, b, e); i = b; j = e - 1
3     while True:
```

```
4           while A[i] >= p:
5               i += 1
6           while A[j] <= p:
7               j -= 1
8           if (j > i):
9               swap(A, i, j)
10          else:
11              return i + (b == i)
```

On va maintenant optimiser la recursion

Appelle recursif terminal:

> C'est lorsqu'on return directement un appelle a une fonction.

```
1  def fac_rec(n, res):
2    if (n <= 1)
3      return res
4    return fac_rec(n - 1, res * n)
```

Le compilateur va donc faire de la magie pour directeemnt sauter dans l'appelle recursif. Ca revient a une boucle. Ce ne sont plus des appelles recursifs meme si on les codes.

```
1  def QuickSort(A, b, e):
2      if (e - b > 1):
3          m = Partition(A, b, e)
4          QuickSort(A, b, m) // Recursion non terminal
5          QuickSort(A, m, e) // Recursion terminal
```

Le dernier `QuickSort` s'ecrit en assembleur:

```
1  QuickSort (A, b, e):
2    if e - b  > 1:
3      m = partition(A,b,e)
4      QuickSort(A,b,m)
5      b = m
6      goto start
```

Du coup dans notre code le compilateur optimise le cote droit de l'arbre des recursions. Mais on peut choisir de quel cote le compilateur va opti. Et on a envi qu'il optimise le cote de l'arbre le plus profond.

```
1  def QuickSort(A, b, e):
```

```
2     while (e - b > 1):
3         m = Partition(A, b, e)
4     if (m - b < e - m):
5       QuickSort(A, b, m) // Recursion non terminal
6       b = m
7     else:
8       QuickSort(A, m, e) // Recursion terminal
9       e = m
```

$\Longrightarrow$ il n'y a pas plus de $\lfloor \log_2 n \rfloor$ appels recursifs empiles.

Avant $S_{QS}(n) = \mathcal{O}(n)$ maintenant $S_{QS}(n) = \mathcal{O}(\log n)$. On a rient gagner en temps mais on a gagner en memoire.

Info: *gcc* lui utilise sa propre pile. Il sait qu'elle sera de taille 32 au max. Du coup il l'aloue en avance et gere tous ces appeles. (C'est chiant a faire).

Petit nombre de valeur

Le `QuickSort` tri de maniere inneficasse les tableaux de petite taille. On peut utiliser `InsertSort` qui lui est efficasse. Il faut trouver le seuil en dessous du quel `InsertSort` devient meilleur que `QuickSort`.

```
1 def QuickSort(A, b, e):
2     if (e - b > TRESHOLD):
3         m = Partition(A, b, e)
4         QuickSort(A, b, m) // Recursion non terminal
5         QuickSort(A, m, e) // Recursion terminal
6   else if e- b > 1:
7     InsertSort(A, b, e)
```

C'est un peut plus chiant a mettre en place dans l'opti precedente. Notement a cause du while. Une des solution serait de faire juste le `Quicksort` puis appeller `InsertSort` sur tout le tableau (Il va faire des decalage de memoire au maximum de TRESHOLD fois).

Dans la *libc* le seuil est a 4. Mais cette valeur a ete trouve sur une vielle machine. Du coup cet valeur est tres dependante des machines.

IntroSort, Tri introspectif

Variante du quicksort, toujours en $\mathcal{O}(n \log n)$.

L'idee est la suivante:

- On fait quicksort
- Si on depasse un seuil disont $c \times \log_2 n$ d'appelles recursif alors on arrete QuickSort et on passe sur un autre tri.

Il va falloir determiner `c` pour optimiser ce tri. Une bonne valeur de `c` est 2.

```
def IntroSort(A, n):
  IntoSort_(A, 0, n, c * log2(n))

def IntroSort_(A, b, e, d):
  if e - b > 1:
    if d == 0:
      HeapSort(a, b, e)
    else:
      m = Partition(A, b, e)
      d -= 1
      IntroSort_(A, b, m, d)
      IntroSort_(A, m, e, d)
```

On est en train de combiner un algo dont le pire cas est pouri avec un autre algo dont la complexite est pas mal mais les constantes sont lourde.

Stabilite des tris

La stabilite est le fait de ne pas changer l'ordre des en memoire des meme valeurs.

SlectionSort	InsertionSort	MergeSort	HeapSort	QuickSort	IntoSort
Instable	Stable	Stable	Instable	Instable	Instable