2019-2020, Semestre d'automne L3, Licence Sciences et Technologies Université Lyon 1

LIFAP6: Algorithmique, Programmation et Complexité

Chaine Raphaëlle (responsable semestre automne)

E-mail: raphaelle.chaine@liris.cnrs.fr http://liris.cnrs.fr/membres?idn=rchaine Abandonnons un peu les Types Abstraits et les Séquences pour revenir aux algorithmes de Tri ©

La foire aux tris

- Les tris internes, sur place, directs et par comparaison
 - Tris élémentaires en $\theta(n^2)$
 - Tri sélection du minimum
 - Tri insertion
 - Tri par permutation ou tri à bulle
 - Mais aussi
 - Tri par tas en θ(nlg₂n) : perfectionnement du tri par sélection
 - Tri par partition (ou tri rapide « quick-sort »): perfectionnement du tri par permutation

Tri par partition (quicksort)

• Idée :

- Partition du tableau à trier autour d'une valeur donnée appelée pivot
- T : tableau[1..n] de Element
- Qu'est-ce que le pivot?
 - Valeur d'un des éléments du tableau à trier

1	2	3	4	5	6	7	8	9	10
124	56	202	26	10	100	45	9	55	3

 Cherchons à mettre les valeurs plus petites que le pivot à gauche et les valeurs plus grande à droite!

Tri par partition (quicksort)

Idée :

- Partition du tableau à trier autour d'une valeur donnée appelée pivot
- T : tableau[1..n] de Element
- Qu'est-ce que le pivot?
 - Valeur d'un des éléments du tableau à trier (sentinelle)
- But de la partition avec sentinelle
 - Effectuer des permutations dans le tableau, de telle sorte que à la fin
 - II existe k1 et k2 t.que
 - T[t]≤pivot pour $1 \le t \le k1$
 - T[t]≥pivot pour k2 ≤ t ≤ n
 - T[t]=pivot pour k1<t<k2</p>
 - Le tableau se retrouve alors divisé en 2 (+1) parties qui peuvent être triées séparément

[1 .. k1]

• Il s'agit donc d'une approche « diviser pour régner »

[k2 .. N]

- Réalisation de la partition avec sentinelle
 - En partant de la gauche du tableau, on recherche le premier élément M ≥ pivot
 - En partant de la droite, on recherche le premier élément m ≤ pivot
 - On permute m et M SI les balayages ne se sont pas croisés. Cela correspond à supprimer une inversion du tableau ou à échanger un élément avec lui-même

2	24	8	35	1	5	15	9	78	20

Exemple (Pivot choisi 15)

- et on poursuit ainsi le double balayage du tableau...
- Remarque : le pivot sert de sentinelle dans la recherche des éléments

- (k2,k1) sont initialisés aux indices de début et de fin du tableau respectivement
- Ils se positionnent sur les indices du premier couple (M,m) formant une inversion
- Attention : après chaque échange servant à éliminer une inversion, k2 et k1 avancent d'une position dans leur direction de balayage
- Puis on recommence....
- Jusqu'à ce que k2 et k1 se croisent

Résultat final :

- 1^{er} cas : trois parties au final (pivot échangé avec lui-même)
- Pivot choisi 15

42	60	10	15	78	1	3	66

- 2^{ème} cas : deux parties au final
- Pivot choisi 15

42	60	10	15	78	100	3	66

Résultat final :

- 1^{er} cas : trois parties au final (pivot échangé avec lui-même)
- Pivot choisi 15

42	60	10	15	78	1	3	66
3	1	10	15	78	60	42	66
		k1		k2			

- 2^{ème} cas : deux parties au final
- Pivot choisi 15

42	60	10	15	78	100	3	66
3	15	10	60	78	100	42	66

k1 k2

Algorithme : procédure partition(donnée-résultat T: tableau[1..n] de Element, donnée pivot : Element, début, fin : 1..n **résultat** : k1, k2 : 1..n) début k2←début, k1←fin répéter tant que T[k2]<pivot faire k2++ fintantque tant que T[k1]>pivot faire k1-fintantque si k2≤k1 alors permutation(T[k1],T[k2]) k2++,k1-finsi jusque k2>k1 (k1 et k2 se sont croisés) fin

Algorithme :

```
procédure triPartition(donnée-résultat T: tableau[1..n] de Element,
                        donnée : début, fin : 1..n)
variables k1,k2 :1..n, pivot : Element
début
  pivot←T[(début+fin)/2] // ... ou autre chose ☺
 partition(T, pivot, début, fin, k1, k2)
  si début<k1 faire
     triPartition(T,début,k1)
 finsi
                                        Données
  si k2<fin faire
     triPartition(T]k2,fin
                                        Résultat
 finsi
                                         Données - Résultat
fin
```

Complexité asymptotique

- Complexité au mieux
 - Si, à chaque récursion, le tableau est coupé en 2 parties gauche et droite de taille égale (cas où le pivot correspond à la médiane)
 - On a alors
 - T(n)=2T(n/2)+T(partition,n)
 or T(partition,n)=θ(n)
 - Donc T(n)=θ(nlg₂n)
 (cas 2 du théorème, avec a=2 et b=2)

Complexité asymptotique

- Complexité au pire
 - Si, à chaque récursion, la partie gauche ou droite est vide (cas où le pivot correspond au plus petit ou au plus grand élément)
 - On a alors
 - T(n)=T(n-1)+T(partition,n)
 or T(partition,n)=θ(n)
 - Donc $T(n)=\theta(n^2)$

```
(T(n)=T(1)+T(partition,1)+T(partition,2)+...+T(partition,n))
```

Choix du pivot

- L'idéal serait de choisir la médiane à chaque partition : peut être coûteux si fait sans soin!
- Un choix fréquent consiste à prendre la médiane
 - du premier élément,
 - de l'élément milieu
 - et du dernier élément de la partie du tableau à partitionner

2	24	8	35	1	5	15	9	78	20

Entre récursif et itératif

- ... mon cœur balance!
- Pour résoudre un même problème, vous avez pu constater que l'on vous présentait souvent une solution récursive et une solution itérative.
- Puissance expressive certaine des solutions récursives :
 - simples et naturelles,
 - stratégie « diviser pour régner »
 - mais il faut savoir les utiliser à bon escient!

- Dérécursification d'un algorithme
 - très intéressante si elle permet d'éviter la répétition de certains calculs
 - permet alors de faire chuter la complexité en temps de calcul d'une solution
- Exemple vus en TD
 - Calcul des termes de la suite de Fibonacci
 - version récursive : profondeur de récursion = n, complexité $\theta(2^n)$
 - version itérative simple : complexité θ(n)
 (il existe également une version itérative utilisant des propriétés mathématiques : complexité θ(lg₂(n)))
- Ces dérécursifications sont liées à la spécificité du problème traité

- Le problème des tours de Hanoï, en revanche, ne pourra pas connaître de solution avec un nombre de déplacements de disques inférieur à 2ⁿ-1
- A complexité équivalente, la dérécursification peut permettre une réduction de l'occupation mémoire occasionnée par la pile des appels récursifs
 - Exemple des versions itératives et récursives de affichageListe décortiqué en LIFAP3
 - Version itérative nécessitant uniquement un pointeur de travail pour « se promener » sur les cellules
 - Version utilisant une procédure interne récursive d'affichage à partir d'une cellule (avec affichage à partir de la première cellule appelant l'affichage à partir de la deuxième, et ainsi de suite...)

```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

main
Pile des appels

x 3
Pile

```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

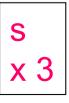
main
Pile des appels



```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=3) main
```

Pile des appels

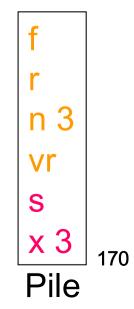


Pile

169

```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

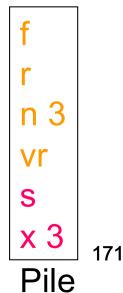
```
fact(n=3)
main
Pile des appels
```



```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=2)
fact(n=3)
main
```

Pile des appels



```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=2)
fact(n=3)
main
```

```
Vr
vr
S
```

```
int fact(int n)
   int f,r;
   if (n \le 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
}
int main()
   int x=3;
                                        fact(n=1)
   int s;
                                        fact(n=2)
   s = fact(x);
                                        fact(n=3)
   return 0;
                                        main
                                   Pile des appels
```

```
vr
vr
S
```

173

```
int fact(int n)
   int f,r;
   if (n \le 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
                                         fact(n=1)
   int s;
                                         fact(n=2)
   s = fact(x);
                                         fact(n=3)
   return 0;
                                         main
                                    Pile des appels
```

```
n
vr
n 2
Vr
n 3
vr
S
Pile
```

```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
                                          fact(n=2)
   s = fact(x);
                                          fact(n=3)
   return 0;
                                           main
```

vr 1 vr vr S 175

```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=2)
fact(n=3)
main
```

Pile des appels



```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=2)
fact(n=3)
main
```

Pile des appels

```
vr
vr
S
```

177

```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
                                           fact(n=2)
   s = fact(x);
                                           fact(n=3)
   return 0;
                                            main
```

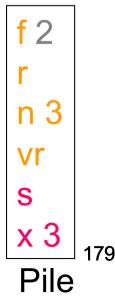
vr 2 vr S

Pile

178

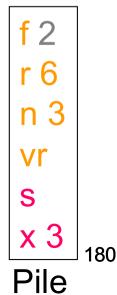
```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=3) main
```



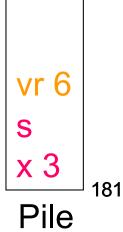
```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=3) main
```



```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

```
fact(n=3)
main
```



```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

main
Pile des appels

s 6 x 3

182

Pile

```
int fact(int n)
   int f,r;
   if ( n <= 1) return 1;
   f = fact(n-1);
   r = n*f;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

main
Pile des appels



Pile

Occupation mémoire beaucoup plus importante que la version itérative!

NEANMOINS LA COMPLEXITE ASYMPTOTIQUE EN TEMPS D'EXECUTION EST IDENTIQUE!

```
int fact(int n)
   int r=1,i;
   for(i=2;i<n+1;i++)
          r = r^*i;
   return r;
int main()
   int x=3;
   int s;
   s = fact(x);
   return 0;
```

fact(n=3) main



- Il existe des appels récursifs facile à dérécursifier :
 - le compilateur peut même le faire pour vous (option d'optimisation)
- Appel récursif terminal :
 - Un appel récursif dont l'exécution précède la sortie de la fonction ou de la procédure appelante
 - Exo : Y a-t-il des appels récursifs terminaux dans triFusionRec?

```
procédure Proc(x : Entier)
début

K(x)

si C(x) alors

J(x), Proc(T(x))

sinon

I(x)

finsi

fin
```

Version itérative par élimination de l'appel récursif terminal?

```
procédure Proc(x : Entier)
début

K(x)

si C(x) alors

J(x), Proc(T(x))

sinon

l(x)

finsi
```

Stratégie d'élimination de l'appel récursif terminal

```
procédure Proc(x : Entier)
début
K(x)
tant que C(x) faire
J(x), x← T(x), K(x)
fintantque
I(x)
fin
```

 $x \leftarrow T(x)$ signifie qu'on recommence le traitement sur T(x)

```
Entier fonction F(x: Entier)

début

K(x)

si C(x) alors

J(x) résultat F(T(x))

sinon

l(x) résultat R1(x)

finsi

fin
```

```
Entier fonction F(x:Entier)

début

K(x)

si C(x) alors

J(x) résultat F(T(x))

sinon

l(x) résultat R1(x)

finsi

fin
```

Stratégie d'élimination de l'appel récursif terminal

```
Entier fonction F(x : Entier)
début
K(x)
tant que C(x) faire
J(x), x 	 T(x), K(x)
fintantque
I(x) résultat R1(x)
fin
```

- Quid de la dérécursification des appels récursifs non terminaux?
- En particulier, que faire si on programme avec un langage ne supportant pas la gestion de la récursivité? (sans grand intérêt sinon...)

Solution :

-Gestion par le programmeur des empilements de paramètres et de valeurs de retour, en utilisant des instances du **type abstrait pile**

module Pile

– importer

Module Element

– exporter

```
Type Pile

procédure initialiser(Résultat p : Pile)
{Préc° : p- non initialisé , Postc° : p+ pile vide}

procédure testament(Donnée-résultat p : Pile)
{Préc° : p- initialisé , Postc° : p+ prêt à disparaitre}

Element fonction consultersommet (p : Pile)
{Préc° : p initialisé et non vide , Résultat : dernier élément empilé}

procédure empile(Donnée-résultat p : Pile, donnée e : Element)
{Préc° : p- initialisé , Postc° : sommet(p+)=e}

procédure dépile(Donnée-résultat p : Pile)
{Préc° : p- initialisé , Postc° : p-= empile(p+,e)}

booléen fonction testVide(p : Pile)
{Préc° : p initialisé , Résultat: vrai si p vide}
```

implantation

 Définitions des éléments offerts par nom_module (cachées à l'utilisateur)

finmodule

- Dérécursification :
 - Empilements de paramètres et de valeurs de retour
 - -II faut également indiquer où on se positionne dans le fil des instructions lorsque l'on a finit le traitement sur des données et que l'on se retrouve dans le traitement de données empilées précédemment!

```
procedure Proc(x)
 debut
   si non CasArret (x)
     T1(x)
    Proc(R1(x))
    T2(x)
     Proc(R2(x))
    T3(x)
   finsi
```

194

```
Procédure Proc (x : paramètre)
Variables
  i : indice, q : paramètre
Début
  Initialisation de la pile P
   Empile(P, \langle x, 1 \rangle)
  // paramètres x de l'appel initial,
  // + Traitement à faire dessus T1
 Tant que P non vide
    <q,i> <- sommet(P), dépile(P)
    //Dépile Paramètres q + traitement à faire dessus T<sub>i</sub>
   T_i(q) // on applique T_i à q
    Empile(P, <q,i+1>) // II nous restera à faire T_{i+1} sur q
    Empile(P, \langle R_i(q), 1 \rangle) // mais avant il faut s'occuper de R_i(q)
  Fin tant que
```

Fin

195