

LIFAP6: Algorithmique, Programmation et Complexité

Chaîne Raphaëlle (responsable semestre automne)
E-mail : raphaelle.chaine@iris.cnrs.fr
<http://iris.cnrs.fr/membres?idn=rchaine>

1

- Abandonnons un peu les Types Abstraits et les Séquences pour revenir aux algorithmes de Tri ☺

151

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 $\theta(n \lg_2 n)$: perfectionnement du tri par sélection
 - Tri par partition (ou tri rapide « *quick-sort* ») : perfectionnement du tri par permutation

152

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!

153

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
 - Il existe k_1 et k_2 t.que
 - $T[t] = \text{pivot}$ pour $1 \leq t \leq k_1$
 - $T[t] = \text{pivot}$ pour $k_2 \leq t \leq n$
 - $T[t] = \text{pivot}$ pour $k_1 < t < k_2$



- Le tableau se retrouve alors divisé en 2 (+1) parties qui peuvent être triées séparément
- Il s'agit donc d'une approche « diviser pour régner »

154

- Réalisation de la partition avec sentinelle
 - En partant de la gauche du tableau, on recherche le premier élément $M \geq \text{pivot}$
 - En partant de la droite, on recherche le premier élément $m \leq \text{pivot}$
 - 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

155

- (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

156

- 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
----	----	----	----	----	-----	---	----

157

- 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

158

- 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++
    fintant que
      tant que T[k1] > pivot faire
        k1--
    fintant que
      si k2 ≤ k1 alors
        permutation(T[k1], T[k2])
        k2++, k1--
  finssi
  jusque k2 > k1 (k1 et k2 se sont croisés)
fin
      
```

159

- 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)
  finssi
  si k2 < fin faire
    triPartition(T, k2, fin)
  finssi
fin
      
```

■ Données

□ Résultat

□ Données - Résultat

160

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
 - $T(n) = 2T(n/2) + T(\text{partition}, n)$
 - or $T(\text{partition}, n) = \theta(n)$
 - Donc $T(n) = \theta(n \lg_2 n)$ (cas 2 du théorème, avec $a=2$ et $b=2$)

161

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
 - $T(n)=T(n-1)+T(\text{partition},n)$
 - or $T(\text{partition},n)=\theta(n)$
 - Donc $T(n)=\theta(n^2)$

$$(T(n)=T(1)+T(\text{partition},1)+T(\text{partition},2)+\dots+T(\text{partition},n))$$

162

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
---	----	---	----	---	---	----	---	----	----

Ici cela correspond au choix de pivot = 2

163

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!

164

- 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 $\theta(n)$ (il existe également une version itérative utilisant des propriétés mathématiques : complexité $\theta(\lg_2(n))$)
- Ces dérécursifications sont liées à la **spécificité du problème traité**

165

- Le problème des tours de Hanoi, en revanche, ne pourra pas connaître de solution avec un nombre de déplacements de disques inférieur à 2^n-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...)

166

Face cachée de la récursivité

```
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

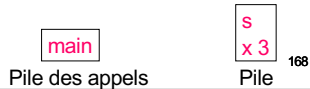
x 3
Pile

167

Face cachée de la récursivité

```
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;
}
```



Face cachée de la récursivité

```
int fact(int n)
{
  int f,r;
  if ( n <= 1) return 1;
  f = fact(n-1);
  r = n*f;
  return r;
}

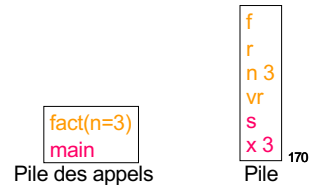
int main()
{
  int x=3;
  int s;
  s = fact(x);
  return 0;
}
```



Face cachée de la récursivité

```
int fact(int n)
{
  int f,r;
  if ( n <= 1) return 1;
  f = fact(n-1);
  r = n*f;
  return r;
}

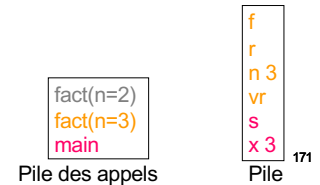
int main()
{
  int x=3;
  int s;
  s = fact(x);
  return 0;
}
```



Face cachée de la récursivité

```
int fact(int n)
{
  int f,r;
  if ( n <= 1) return 1;
  f = fact(n-1);
  r = n*f;
  return r;
}

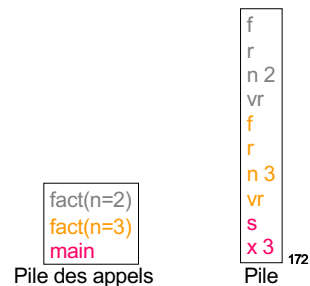
int main()
{
  int x=3;
  int s;
  s = fact(x);
  return 0;
}
```



Face cachée de la récursivité

```
int fact(int n)
{
  int f,r;
  if ( n <= 1) return 1;
  f = fact(n-1);
  r = n*f;
  return r;
}

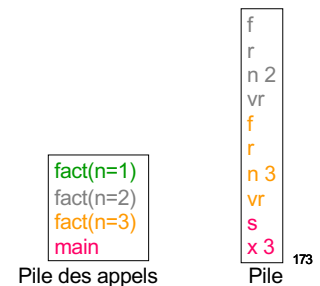
int main()
{
  int x=3;
  int s;
  s = fact(x);
  return 0;
}
```



Face cachée de la récursivité

```
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;
}
```



Face cachée de la récursivité

```
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=1)
fact(n=2)
fact(n=3)
main

Pile des appels

f
r
n 1
vr
f
r
n 2
vr
f
r
n 3
vr
s
x 3

Pile 174

Face cachée de la récursivité

```
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 1
f
r
n 2
vr
f
r
n 3
vr
s
x 3

Pile 175

Face cachée de la récursivité

```
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

f 1
r
n 2
vr
f
r
n 3
vr
s
x 3

Pile 176

Face cachée de la récursivité

```
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

f 1
r 2
n 2
vr
f
r
n 3
vr
s
x 3

Pile 177

Face cachée de la récursivité

```
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 2
f
r
n 3
vr
s
x 3

Pile 178

Face cachée de la récursivité

```
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

f 2
r
n 3
vr
s
x 3

Pile 179

Face cachée de la récursivité

```
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

f 2
r 6
n 3
vr
s
x 3

Pile 180

Face cachée de la récursivité

```
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

vr 6
s
x 3

Pile 181

Face cachée de la récursivité

```
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

Pile 182

Face cachée de la récursivité

```
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

0

Pile 183

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

Pile des appels

i
r
n
vr
s
x 3

Pile 184

Récursivité terminale

- Il existe des appels récursifs facile à dérécurifier :
 - 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?

185

Réversivité terminale

```

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?

186

Réversivité terminale

```

procédure Proc(x : Entier)
début
  K(x)
  si C(x) alors
    J(x), Proc( T(x) )
  sinon
    I(x)
  finsi
fin
    
```

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 $T(x)$

187

Réversivité terminale

```

Entier fonction F(x : Entier)
début
  K(x)
  si C(x) alors
    J(x) résultat F( T(x) )
  sinon
    I(x) résultat R1(x)
  finsi
fin
    
```

188

Réversivité terminale

```

Entier fonction F(x : Entier)
début
  K(x)
  si C(x) alors
    J(x) résultat F( T(x) )
  sinon
    I(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
    
```

189

- 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...)

190

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

191

```

• 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 disparaître}
    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
- Il faut également indiquer où on se positionne dans le fil des instructions lorsque l'on a fini le traitement sur des données et que l'on se retrouve dans le traitement de données empilées précédemment!

```

procédure Proc(x)
  debut
    si non CasArret (x)
      T1(x)
      Proc(R1(x))
      T2(x)
      Proc(R2(x))
      T3(x)
    finsi
    ....
  fin

```

```

Procédure Proc (x : paramètre)
Variables
  i : indice, q : paramètre
Début
  Initialisation de la pile P
  Empile(P, <x, 1>)
  // 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 Ti
    Ti(q) // on applique Ti à q
    Empile(P, <q,i+1>) // Il nous restera à faire Ti+1 sur q
    Empile(P, <Ri(q),1>) // mais avant il faut s'occuper de Ri(q)
  Fin tant que
Fin

```