2024-2025, Semestre d'automne
L3, Licence Sciences et Technologies
Université Lyon 1

LIFAPC: Algorithmique, Programmation et Complexité

Chaine Raphaëlle (responsable semestre automne)
E-mail : raphaelle.chaine@liris.cnrs.fr
http://liris.cnrs.fr/membres?idn=rchaine

1

1

Spécificités des algorithmes itératifs et récursifs

- Outils de preuve
  - Algorithmes itératifs :
    - **Assertion** : propriété que vous pensez être vraie en un point du programme (utilisation de `assert`)
    - Instructions de branchement devant prendre en compte tous les cas de figure possible
    - Invariant de boucle permettant de **progresser vers une condition d'arrêt** (raisonnement par récurrence)
    - Satisfaction d'une assertion au moment où la condition d'arrêt est atteinte
  - Algorithmes récursifs
    - Raisonnement par récurrence

124

124

- **Exemple de preuve d'un algorithme itératif**
- **procédure** rechercheDansSequence(
  donnée s : séquence, e: element,
  résultat i : position, b :booléen)
  {précondition : s, e initialisés
  postcondition : si b contient vrai
  alors e est en ième position de s,
  sinon e n'est pas dans s}
  début
  i←1, b ←faux
  tantque (i ≤ taille(s)) **etalors** (e≠s[i]) **faire**
  i++
  fintantque
  si i ≤ taille(s) **alors**
  b←true
  finsi
  fin

125

125

- Invariant de boucle
- i←1, b ←faux
  tantque (i ≤ taille(s)) **etalors** (e≠s[i]) **faire**
  {Assertion : Au kième passage éventuel dans la boucle, k=i et pour tout j t.que 1 ≤ j < k on a s[j] ≠ e}
  i++
  fintantque
- Preuve de l'invariant de boucle
- Récurrence
  - Vrai pour k=1
  - Supposons le vrai pour k=K
  - Si on passe une K+1 ième fois dans la boucle, cela signifie que K+1 ≤ taille(s) et que s[K+1] ≠ e donc s[j] ≠ e pour tout 1 ≤ j < K+1, ce qui clôt la récurrence

126

126

- Il faut également montrer que le corps de la boucle permet de progresser vers la condition d'arrêt : pas de boucle infinie!
  - Au K^{ième} passage dans la boucle, i augmente strictement : on ne pourra pas passer une infinité de fois dans la boucle, puisqu'une des conditions d'arrêt sera atteinte quand i excèdera taille(s)
- A la sortie de la boucle :
  - 1er cas : si i> taille(s) alors l'invariant du dernier passage dans la boucle assure que pour 1 ≤ j ≤ taille(s) on a s[j] ≠ e, donc e n'est pas dans s, ce qui est cohérent avec le fait que b contienne faux
  - 2ème cas : si i ≤ taille(s) alors s[i] vaut e ce qui est cohérent avec l'affectation de vrai à b

127

127

- **Exemple de preuve d'un algorithme récursif**
- Tri fusion d'un tableau
- **procédure** TriFusionRec(
  donnée-résultat tab : tableau[1..n] d'Elements,
  données p, r : 1..n)
  {précondition : aucune,
  postcondition : tab trié entre les indices p et r}
  variable
  q : 1..n
  début
  si p<r **alors**
  q←(p+r)/2
  TriFusionRec(tab,p,q) tri 1ère moitié
  TriFusionRec(tab,q+1,r) tri 2nde moitié
  Fusionner(tab,p,q,r)
  finsi
  fin

	p			q	q+1			r
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
24	56	2	26	10	100	45	9

128

128

- Raisonnement par récurrence :
  - Pour un(e partie de) tableau de taille 0 ou 1, l'algorithme est correct
  - On suppose que l'algo est correct pour un (sous) tableau de taille m ≥1, montrons qu'il est correct pour un (sous) tableau de taille m+1
    - Comme m+1 ≥ 2 on a p<r et p≤q<r
      les 2 appels récursifs se font donc sur des tableaux de taille inférieure à m+1
    - Sous réserve que la procédure de fusion soit correcte, l'algorithme de tri fusion sera donc correct

129

129

Spécificités des algorithmes itératifs et récursifs

- Calculs de complexité
- Complexité des algorithmes itératifs :
  - Utilisation des règles de complexité des séquences d'instruction (boucles, traitements conditionnels, etc…)
- Complexité des algorithmes récursifs :
  - Solution d'une équation de récurrence

130

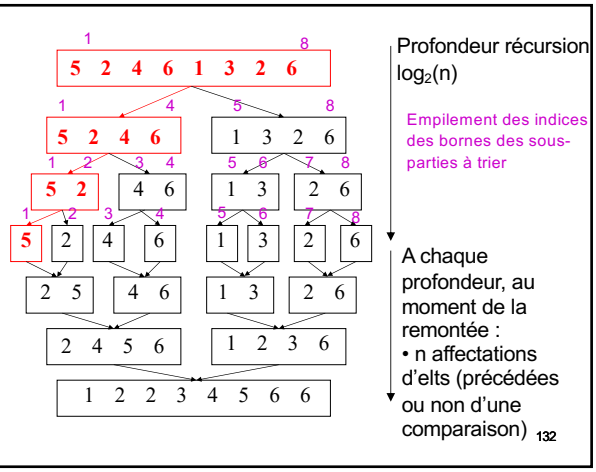
130

Complexité d'un algorithme récursif

- Le temps d'exécution d'un algorithme récursif est généralement solution d'une équation de récurrence
- Exemple : Analyse de la procédure TriFusionRec
  - Appelons $T_{TF}(n)$ le temps d'exécution de l'algorithme sur un tableau de taille n

131

131



132

132

Temps $T_{TF}(n)$ pour trier un tableau de taille n

- $T_{TF}(1)$=T(initialisation des paramètres formels) +T(comparaison d'indice)=C_1
- Pour n≥2
  $T_{TF}(n)$= T(initialisation des paramètres formels)
  + T(comparaison d'indice)
  + T(affectation d'indice)
  + 2*$T_{TF}(n/2)$
  + T(fusionner,n)
- Or T(fusionner,n)=θ(n)

133

133

$$T(n) = \begin{cases} C1 \text{ si } n=1 \\ 2T(n/2)+\theta(n) + C1 + C2 \text{ si } n>1 \end{cases}$$

$$T(n) = \begin{cases} C1 \text{ si } n=1 \\ 2T(n/2)+\theta(n) \text{ si } n>1 \end{cases}$$

134

134

Résolution

Méthode par développement itératif
Soit n>1 et soit k=$lg_2(n)$
(on supposera ici que n correspond exactement à 2^k)
T(n)=2T(n/2)+θ(n)
ie. T(n) = 2(2T(n/2^2)+ θ(n/2)) + θ(n)
ie. T(n) = 2^2T(n/2^2)+ 2θ(n/2) + θ(n)
ie. T(n) = 2^kT(n/2^k)+ 2^{k-1}θ(n/2^{k-1}) + 2^{k-2}θ(n/2^{k-2}) … +θ(n)
ie. T(n) = 2^kT(n/2^k)+ θ(n) + … +θ(n)
ie. T(n) = n C1 + $lg_2(n)$θ(n)
ie. T(n) = θ(n $lg_2(n)$)

135

135

- Il s'agit du temps d'exécution d'un algorithme qui divise un problème de taille n en a≥1 sous-problèmes de taille n/b avec b>1
(stratégie **« diviser pour régner »**)

$$T(n) = aT(\frac{n}{b}) + f(n) \qquad n>0$$

f(n) décrit le coût de la division du problème en a sous-problèmes et de la recomposition des résultats

136

136

Master Theorem

1. Si il existe $\varepsilon > 0$ t. que $f(n) = O(n^{\log_b(a)-\varepsilon})$ alors $T(n) = \Theta(n^{\log_b(a)})$
2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \log_b(n))$
3. Si il existe $\varepsilon > 0$ t.que $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ et il existe $c < 1$ t. que $af(\frac{n}{b}) \le cf(n)$ pour n grand alors $T(n) = \Theta(f(n))$

Intuition de preuve au tableau

137

137

Master Theorem

1. Si il existe $\varepsilon > 0$ t. que $f(n) = O(n^{\log_b(a)-\varepsilon})$ alors $T(n) = \Theta(n^{\log_b(a)})$ Coût de la décomposition et recomposition négligeable devant le coût lié à la récursion
2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \log_b(n))$
Coût de la décomposition et recomposition similaire à celui lié à la récursion
3. Si il existe $\varepsilon > 0$ t.que $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ et il existe $c < 1$ t. que $af(\frac{n}{b}) \le cf(n)$ pour n grand alors $T(n) = \Theta(f(n))$

Coût de la décomposition et recomposition non négligeable devant le coût lié à la récursion, … mais maîtrisé!

138

138

- Construisons ensemble une procédure récursive dont la complexité augmente avec la valeur d'un entier n passé en paramètre.
- On souhaite connaître son comportement asymptotique quand n augmente.
- Cette fonction contient des instructions d'affichage (« Coucou ») sur la sortie standard et répond à une stratégie récursive.
  - Cette fonction s'appelle elle-même 6 fois (a=6) pour une valeur de n divisée par b=3.
  - Les instructions d'affichage sont situées en dehors du cas d'arrêt

139

139

```
void proc(int n)
 { if(n>0)
  {for(int i=0;i<6;i++)
        proc(n/3);
     for(int j=0;j<n;j++)
       affiche(« coucou »);
  }
}
```

- Application du Master Theorem
  - f(n)=n affichages de « Coucou » hors des appels récursifs emboîtés.
  - $log_3(6)$ = 1.6309… (ie de la forme 1+epsilon)
  - Cas 1 du Master Theorem : T(n)=θ($n^{log3(6)}$)

140

140