Programmation Fonctionnelle
Complexité Algorithmique

Michael Lampis

2023-2024

Complexité et Programmation Fonctionnelle

- Complexité d'un programme : quantité de ressources nécessaires (surtout temps, mais aussi mémoire) pour son exécution, **dans le pire de cas**, comme fonction de la taille n de l'entrée.
  - Dans le pire de cas $\rightarrow$ on veut donner une borne supérieure qui ne sera dépassée pour aucune entrée de taille n.
- Dans le contexte de Haskell, plusieurs questions se posent :
  - Comment analyser la complexité d'un programme ?
  - Comment l'améliorer sans changer son résultat ?
- Dans ces slides :
  - On parle rapidement de l'analyse de complexité (en prenant en compte l'évaluation paresseuse)
  - On présente quelques techniques pour améliorer la complexité de programmes récursifs.

Analyse de complexité

Contexte : étant donné un programme de Haskell, estimer sa complexité.

- Deux difficultés principales :
  - Fonctions récursives
  - Évaluation Paresseuse

Fonction récursives

- L'analyse de la complexité de fonctions récursive se base sur la résolution de relations de récurrence.

```
mergesort [] = []
mergesort [x] = [x]
mergesort xs = merge l1 l2
 where
  l1 = mergesort (take (length xs `div` 2) xs)
  l2 = mergesort (drop (length xs `div` 2) xs)
merge [] xs = xs
merge xs [] = xs
merge (x:xs) (y:ys)
 | x<y = x:merge xs (y:ys)
 | otherwise = y:merge (x:xs) ys
```

Fonction récursives

- L'analyse de la complexité de fonctions récursive se base sur la résolution de relations de récurrence.
- `merge` a complexité $T_1(n) = T_1(n-1) + O(1)$ ou n est la somme de tailles de deux listes données
  - Chaque appel diminue une de deux listes par un élément.
  - Chaque appel a un coût constant ($O(1)$), sous la supposition que `(<), (:)` prennent de temps constant.
- `mergesort` a complexité $T_2(n) = 2T_2(n/2) + T_1(n) + O(1)$
  - Selon le Master Theorem cela donne $T_2(n) = O(n \log n)$.

Fonction récursives – Exemple 2

- Quelle est la complexité de la fonction suivante ?

```
fibo 1 = 1
fibo 2 = 1
fibo n = fibo (n-1) + fibo (n-2)
```

- $T(n) = T(n-1) + T(n-2)$
- $T(1) = T(2) = 1$
- Cette relation est exactement la définition de la suite de Fibonacci !
- $\rightarrow T(n) = O(F_n)$
- Pour résoudre une telle relation on “devine” que $T(n)$ a la forme r^n
- $\rightarrow r^n = r^{n-1} + r^{n-2}$
- $\rightarrow r^2 = r + 1$
- $\rightarrow r \approx 1.618$ (nombre d’or)
- $T(n) = O(1.619^n)$
- **Complexité Exponentielle !!**

Lazy evaluation

- Haskell n'évalue une expression que si on en a besoin.
- Les instructions du type `x = expr` n'ont aucun effet immédiat !
- Exemple :

```
f n = 1+f n
g n = 5
```

Exécution :

```
*Main> map (\x -> (g x,f x)) [1..5]
[(5, -- on bloque ici ! f a boucle infinie... )]
*Main> map fst ( map (\x -> (g x,f x)) [1..5] )
[5,5,5,5,5] -- on n'a pas eu a calculer f !
(0.02 secs, 81,464 bytes)
```

Être paresseux – Pour ou Contre ?

- Avantages de l'évaluation paresseuse
  - Augmente la probabilité que le programme termine correctement.
  - Permet d'utiliser des structures infinies.
  - Ne calcule pas les parties non–nécessaires $\rightarrow$ peut être plus rapide.
- Désavantages
  - Prédire où (et pourquoi) un programme bloque devient beaucoup plus compliqué.
  - Compilateur est obligé de conserver des parties non–évaluées du programme (au cas où on en aura besoin) $\rightarrow$ manque d'efficacité
  - Augmente la probabilité qu'on calcule la même chose plusieurs fois.
- Leçon : il faut être conscient du comportement paresseux de Haskell, et faire en sorte que ça ne nous oblige pas de répéter des calculs.

Optimisation en Haskell

- Le but principal de ces slides et d'expliquer les pièges de complexité associés avec la programmation récursive (et l'évaluation paresseuse) en Haskell, et expliquer comment les contourner.

Exemple :

```
fibo 1 = 1
fibo 2 = 1
fibo n = fibo (n-1) + fibo (n-2)
```

- Complexité exponentielle !?
- La source de notre problème est que ce programme va calculer chaque valeur F_k, F_{n-k} fois, alors qu'une seule fois aurait suffit !

```
fibo n=fibo (n-1)+fibo (n-2)
      =fibo (n-2)+fibo (n-3)+fibo (n-3)+fibo (n-4)
      =fibo (n-3)+fibo (n-4)+fibo (n-4)+fibo (n-5)+...
```

Fast Fibonacci

- Problème : le compilateur n'est pas assez intelligent pour constater que `fibo (n-3)` est calculé plusieurs fois.
  - Or, grâce à la transparence référentielle, on sait que calculer chaque valeur intermédiaire une seule fois suffit (cette valeur ne peut pas changer).
- Pour éviter l'explosion combinatoire, il faut donner une instruction explicite de stocker les résultats intermédiaires.
  - Problème sensible, car, à cause de l'évaluation paresseuse, l'opérateur = ne fait pas du stockage (comme en C).
- Technique générale : on déclare dans l'intérieur de notre fonction récursive...
  - Une **liste** qui stocke les résultats intermédiaires
  - Une fonction auxiliaire qui fait un pas de l'appel récursif et utilise la liste pour les valeurs précédentes.

Fast Fibonacci

```
fibo n = f n
 where
   f 0 = 1
   f 1 = 1
   f n = fibs !! (n-1) + fibs !! (n-2)
   fibs = [ f x | x<-[0..n-1] ]
```

Explication :

- On a déclaré une fonction interne `f`
  - Idée : `f == fibo` mais sans appel récursif
  - Elle va fonctionner sous la supposition que la liste `fibs` a stocké les valeurs correctes de `f` (donc de `fibo`) pour des arguments inférieurs.
- On a aussi déclaré une liste `fibs`
  - Cette liste stocke toutes les valeurs intermédiaires
  - Assertion : `fibs !! i == fibo i`

Fast Fibonacci

```
fibo n = f n
 where
  f 0 = 1
  f 1 = 1
  f n = fibs !! (n-1) + fibs !! (n-2)
  fibs = [ f x | x<-[0..n-1] ]
```

Preuve de correction :

- Assertions : $\forall i$:
  - `fibo i == f i`
  - `fibs !! i == fibo i`
- Induction : Assertions vraies pour $i = 0, 1$
- Si assertions vraies pour $i - 1, i - 2$
  - `f i = fibs !! (i-1) + fibs !! (i-2)`
  - `== fibo (i-1) + fibo (i-2) == fibo i`

Fast Fibonacci

```
fibo n = f n
 where
   f 0 = 1
   f 1 = 1
   f n = fibs !! (n-1) + fibs !! (n-2)
   fibs = [ f x | x<-[0..n-1] ]
```

Complexité :

- La complexité de `f n` est $O(n)$
  - L'opérateur `(!!)` prend du temps $O(n)$.
- Pour calculer `fibs` on paie $\sum_{i=1}^{n} O(i) = O(n^2)$
- Donc, complexité $O(n^2)$ (quadratique)
- Déjà une énorme amélioration !
- Pour atteindre $O(n)$ il faut utiliser à la place de la liste une structure qui permet d'accéder à ses éléments en $O(1)$ (`Array`).

Leçons

- En Haskell, c'est très naturel d'écrire des fonctions récursives
- C'est trop facile d'arriver à une complexité exponentielle !
- Souvent (mais pas toujours !), cette complexité est évitable, si elle est dû à une mauvaise organisation du calcul (répétition des mêmes évaluations).
- Dans ce cas, on utilise une structure de données pour stocker les résultats intermédiaires et on évite de les recalculer.
  - Attn: il faut écrire cette structure en sorte que les valeurs qui sont nécessaires en première sont calculées dans le bon ordre.
  - Exemple : aurait-on pu écrire `fibs = [ f x | x<-[n,n-1..1] ]` dans le programme précédent ?
  - Cela impliquerait, qu'on calcule d'abord `f n`, puis `f (n-1)`,...
  - (En fait, pas exactement, grâce à l'évaluation paresseuse, donc ça pourrait marcher, mais il faut faire attention...)

Nombres Premiers

Pour un autre exemple, considérez le programme suivant :

```
prime n = [ i | i<-[2..n-1], n`mod`i==0 ] == []

allprimes n = filter prime [2..n]
```

- Complexité de la fonction `prime` $= O(n)$
- Complexité de `allprimes` $= O(n^2)$
- `allprimes n` affiche les premiers inférieurs à `n`

```
*Main> allprimes 100
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,7
(0.02 secs, 492,936 bytes)
*Main> sum ( allprimes 50000 )
121013308
(64.98 secs, 33,029,198,848 bytes)
```

- Comment accélérer `allprimes` ?

Crible d'Ératosthène

- Algorithme découvert il y a 2000 ans par un mathématicien grec !
- Idée, à la place de tester chaque entier, on élimine les multiples de chaque entier (qui sont forcément non–premiers)

```
sieve n = s [2..n]
 where
  s [] = []
  s (x:xs) = x: s ( remove xs [2*x, 3*x..] )
  remove [] _ = []
  remove (x:xs) (y:ys)
   | x==y = remove xs ys
   | x<y  = x:remove xs (y:ys)
   | otherwise = remove (x:xs) ys

*Main> sum ( sieve  50000 )
121013308
(7.93 secs, 5,521,567,080 bytes)
```

Crible amélioré

- Pour améliorer la première version, on peut programmer `prime` avec une complexité de $O(\sqrt{n})$ (on teste les diviseurs $\leq \sqrt{n}$).
- On fait une amélioration similaire pour l'algorithme d'Ératosthène

```
sieve n = s [2..n]
 where
  s [] = []
  s (x:xs) = x: s ( remove xs [x*x, x*x+x..n] )
  remove [] _ = []
  remove xs [] = xs
  remove (x:xs) (y:ys)
   | x==y = remove xs ys
   | x<y  = x:remove xs (y:ys)
   | otherwise = remove (x:xs) ys

*Main> sum ( sieve  50000 )
121013308
(0.37 secs, 160,299,496 bytes)
```

Programmation Dynamique

- L'idée de stocker des valeurs intermédiaires pour accélérer un algorithme est liée à une technique algorithmique générique appelée **Programmation Dynamique**.
  - Cette problématique n'est pas restreinte au domaine de la programmation fonctionnelle.
  - Les mêmes problèmes se posent pour les langages impératifs.
- Programmation Dynamique : cas d'usage
  - Problème d'optimisation
  - Algorithme récursif facile à trouver (mais exponentiel !)
  - Si le problème a la propriété de l'optimalité de sous–problèmes
    - La solution optimale est composée des solutions optimales de sous–problèmes.
  - . . . et si le nombre de sous–problèmes à considérer est polynomial
  - On stocke les solutions de tous les sous–problèmes et les utilise pour construire des solutions de super–problèmes de façon inductive.

Exemple – Max Independent Set

- Considérez le problème suivant : étant donné une liste de `Int` on veut sélectionner un sous-ensemble de ses éléments en sorte que :
  - On ne sélectionne pas deux éléments consécutifs
  - La somme des éléments sélectionnés est maximum

Exemples :

```
*Main> maxis [3,5,1,2,9,4]
[5,9]
*Main> maxis [9,1,3,8,2]
[9,8]
*Main> maxis [9,1,3,8,2,7,12,6]
[9,8,7,6]
```

- **NB**: sélectionner l'élément max n'est pas forcément optimal !
- Donner une fonction Haskell pour `maxis`

Max Independent Set – v1

- Deux solutions optimales possibles :
  - On prend le premier élément (et donc le deuxième est interdit) et la meilleure solution à partir du troisième
  - On ne prend pas le premier élément (donc on prend la meilleure solution à partir du deuxième)

```
maxis :: [Int] -> [Int]
maxis [] = []
maxis [x] = [x]
maxis [x,y] = [max x y]
maxis (x:y:xs)
  | x+sum (maxis xs) > sum (maxis (y:xs)) = x:maxis xs
  | otherwise = maxis (y:xs)
```

- Complexité : $T(n) = T(n-1) + T(n-2)$
- Notre vieil ami Fibonacci ! $\rightarrow O(1.6^n)$!!
- Ne fonctionne pas pour $n \geq 40$!?!?

Max Independent Set – v2

- Idée : Stocker dans `ms!!i` la solution optimale pour les `i` premiers éléments de la liste donnée

```
maxis :: [Int] -> [Int]
maxis xs = m (length xs)
 where
  m 0 = []
  m 1 = [head xs]
  m i
   | (xs!!(i-1))+sum (ms!!(i-2))>sum (ms!!(i-1)) =
                              ms!!(i-2) ++ [xs!!(i-1)]
   | otherwise = ms!!(i-1)
  ms = [ m i | i<-[0..length xs] ]
```

- Complexité de `m`: $O(n)$ (à cause du `(!!)`)
- Complexité totale : $O(n^2)$
- Essayez pour $n > 100$ pour vérifier l'amélioration !

Shortest Superlist

- Problème : on est donné deux listes `xs,ys`
- On cherche la plus courte liste qui contient `xs,ys` comme sous–listes
  - `zs` contient `xs` comme sous–liste si `zs` contient tous les éléments de `xs` dans le même ordre (mais pas forcément consécutifs).

Exemples :

```
*Main> minsuperlist "abcabc" "bababababa"
"babcabcaba"
*Main> minsuperlist "abracadabra" "babaaurhum"
"abracbadaburahum"
*Main> minsuperlist [2,4..10] [6,4,8,2]
[2,4,6,4,8,10,2]
```

Shortest Superlist – v1

- Idée récursive : si premier élément de `xs,ys` est le même, très bien ! On commence avec ça et regarde `tail xs, tail ys`.
- Sinon, il faut soit commencer avec le premier élément de `xs`, soit avec celui de `ys`. On choisit la meilleure solution.

```
minsuperlist :: Eq a => [a] -> [a] -> [a]
minsuperlist [] ys = ys
minsuperlist xs [] = xs
minsuperlist (x:xs) (y:ys)
 | x==y       = x:minsuperlist xs ys
 | otherwise = if l1>l2 then sol2 else sol1
 where
  sol1 = x:minsuperlist xs (y:ys)
  sol2 = y:minsuperlist (x:xs) ys
  l1 = length sol1
  l2 = length sol2
```

- Complexité : $O(2^n)$!?
- Aucun espoir si $n \geq 40$...

Shortest Superlist – v2

- On stocke la solution optimale si on considère les i premiers éléments de `xs` et les j premiers éléments de `ys`

```
minsuperlist :: Eq a => [a] -> [a] -> [a]
minsuperlist xs ys = msl (length xs) (length ys)
 where
  msl 0 j = take j ys
  msl i 0 = take i xs
  msl i j
   | (xs!!(i-1)) == (ys!!(j-1)) =
        (msls !! (i-1) !! (j-1)) ++ [xs !!(i-1)]
   | length (msls !!(i-1)!!j)>length (msls !!i!!(j-1))=
       msls!!i!!(j-1)++[ys!!(j-1)]
   | otherwise = msls!!(i-1)!!j++[xs!!(i-1)]
  msls=[[msl i j |j<-[0..length ys]] |i<-[0..length xs]]
```

- Complexité $O(n^3)$ (pourquoi ?)

Structures de Données

Listes vs Tableaux vs Map

- Dans les exemples précédents on a utilisé des listes pour stocker les résultats intermédiaires de nos calculs.
- Or, l'opération `!!` prend de temps linéaire ($O(n)$) pour les listes, ce qui n'est pas très efficace.
  - **NB**: l'amélioration qu'on a faite nous a permis de passer d'une complexité du type 2^n à $O(n^2)$ ou similaire. La question est maintenant si on peut arriver à $O(n)$. Ce dernier pas serait, certes, important, mais la grande amélioration a déjà été faite...
- Solution : utiliser une autre structure de données à la place des listes.

Listes vs Tableaux vs Map

	Insertion (début)	! !	Suppression
Liste	$O(1)$	$O(n)$	$O(n)$
Tableau	$O(n)$	$O(1)$	$O(n)$
Map	$O(\log n)$	$O(\log n)$	$O(\log n)$

- Liste : rapide d'ajouter un élément au début, pas efficace pour accéder à un élément quelconque (il faut traverser la liste)
- Tableau : rapide d'accéder à un élément arbitraire, pas efficace d'ajouter/supprimer des éléments
- Map : Tableau associatif implémenté avec ABR équilibré $\rightarrow$ toutes opérations prennent $O(\log n)$

Tableaux

- `import Data.Array`
- Équivalent de tableaux de C.
- Indices peuvent avoir n'importe quel type de la classe `Ix`
  - Notamment `Int, Integer`
- Indices peuvent former n'importe quel intervalle.
- Pour accéder à un élément on utilise `!`.
- Construire un tableau avec la fonction `array`

```
Prelude Data.Array> :type array
array :: Ix i => (i, i) -> [(i, e)] -> Array i e
```

Arguments :

- (Min, Max) indice
- Liste des associations (indice, valeur)

Tableaux – Exemples

```
> a1 = array (0,9) [ (i,2*i) | i<-[0..9] ]
> a1 ! 5
10
> a2 = array ( (1,1), (3,3) ) [ ((i,j),2*i+j)
                              | i<-[1..3], j<-[1..3] ]
> a2
array ((1,1),(3,3))
  [((1,1),3),((1,2),4),((1,3),5),
   ((2,1),5),((2,2),6),((2,3),7),
   ((3,1),7),((3,2),8),((3,3),9)]
> a2 ! (2,2)
6
> a2 ! (2,7)
*** Exception: Error in array index
```

Fibonacci again

```
import Data.Array

fibo n = f n
 where
  f 0 = 1
  f 1 = 1
  f i = (fibs ! (i-1)) + (fibs ! (i-2))
  fibs = array (0,n) [ (x,f x) | x<-[0..n] ]
```

NB: dernière ligne était auparavant

```
fibs = [ f x | x<-[n,n-1..0] ]
```

- Et alors ?

Fibonacci again

```
import Data.Array

fibo n = f n
 where
  f 0 = 1
  f 1 = 1
  f i = (fibs ! (i-1)) + (fibs ! (i-2))
  fibs = array (0,n) [ (x,f x) | x<-[0..n] ]
```

NB: dernière ligne était auparavant

```
fibs = [ f x | x<-[n,n-1..0] ]
```

- Complexité devient $O(n)$ à la place de $O(n^2)$.

Max Independent Set

```
import Data.Array

maxis :: [Int] -> [Int]
maxis xs = reverse $ fst $ m (length xs)
 where
  xs' = array (0,length xs-1) $ zip [0..] xs
  m 0 = ([], 0)
  m 1 = ([head xs], head xs)
  m i
   | (xs'!(i-1))+(snd (ms!(i-2)))>(snd (ms!(i-1))) =
                          ([xs'!(i-1)] ++ fst (ms!(i-2)),
                           snd (ms!(i-2)) + (xs'!(i-1)))
   | otherwise = ms!(i-1)
  ms = array (0,length xs) [ (i,m i) | i<-[0..length xs] ]
```

Map

- `import Data.Map`
- Tableaux associatifs implémentés avec arbres binaires de recherche **équilibrés**.
- Clés peuvent avoir n'importe quel type de la classe `Ord`
  - Notamment `Int,Integer`
- Pour accéder à un élément on utilise `!`.
- Construire un map avec la fonction `fromList`

```
> :t fromList
fromList :: Ord k => [(k, a)] -> Map k a
```

Argument :

- Liste des associations (clé, valeur)

Max Independent Set – Map

```
import Data.Map

maxis :: [Int] -> [Int]
maxis xs = reverse $ fst $ m (length xs)
 where
  xs' = fromList $ zip [0..] xs
  m 0 = ([], 0)
  m 1 = ([head xs], head xs)
  m i
   | (xs'!(i-1))+(snd (ms!(i-2)))>(snd (ms!(i-1))) =
                          ([xs'!(i-1)] ++ fst (ms!(i-2)) ,
                           snd (ms!(i-2)) + (xs'!(i-1)))
   | otherwise = ms!(i-1)
  ms = fromList [ (i,m i) | i<-[0..length xs] ]
```

Tableaux vs Map

- Opération ! légèrement plus efficace pour les tableaux ($O(1)$ vs. $O(\log n)$)
  - Différence pas très significative : $n < 2^{30}$ pratiquement toujours
- Avantages de Map :
  - Peut utiliser plus de types–clés
  - Clés ne doivent pas forcement former un intervalle
  - Modifications rapides possibles

Exemples

Prefix Sums

Étant donné une liste `xs`, calculer une liste qui contient la somme de chaque préfixe de `xs`.
Exemple :

```
> sumEasy  [1,2,3,4,5]
[1,3,6,10,15]
```

Prefix Sums

Étant donné une liste `xs`, calculer une liste qui contient la somme de chaque préfixe de `xs`.
Exemple :

```
> sumEasy  [1,2,3,4,5]
[1,3,6,10,15]
```

Solution facile :

```
sumEasy :: [Integer] -> [Integer]
sumEasy xs = [ sum $ take i xs | i<-[1..length xs] ]
```

Complexité ?

Prefix Sums

Étant donné une liste `xs`, calculer une liste qui contient la somme de chaque préfixe de `xs`.
Solution avec Tableaux :

```
import Data.Array

sums :: [Integer] -> Array Int Integer
sums xs = mysums
 where
  xsa = array (0,length xs-1) $ zip [0..] xs
  mysums = array (0,length xs-1) $
           zip [0..]
           [ if i==0 then xsa!0 else mysums!(i-1)+xsa!i
                                   | i<-[0..length xs-1] ]
```

Complexité ?

MaxSubArray

Étant donné une liste d'entiers `xs`, trouver un intervalle continu de `xs` avec somme maximum.
Exemples :

```
> maxsubarray [1,2,3,-5,2,3]
[1,2,3,-5,2,3]
> maxsubarray [1,2,3,-8,2,3]
[1,2,3]
> maxsubarray [1,2,3,-8,2,3,4]
[2,3,4]
```

MaxSubArray

Étant donné une liste d'entiers `xs`, trouver un intervalle continu de `xs` avec somme maximum.
Solution facile :

```
maxsubarray :: [Integer] -> [Integer]
maxsubarray xs = take size $ drop left $ xs
 where
 (s,left,size) = maximum [ (sum (take j (drop i xs)),i,j)
                         | i<-[0..length xs-1],
                           j<-[1..length xs-i] ]
```

Complexité ?

MaxSubArray

Étant donné une liste d'entiers `xs`, trouver un intervalle continu de `xs` avec somme maximum.
Solution avec Tableaux :

```
import Data.Array

maxsubarrayS :: [Integer] -> Integer
maxsubarrayS xs = maximum best
 where
  xsa = array (0,length xs-1) $ zip [0..] xs
  cur = array (-1,length xs-1) $ zip [-1..] (0:[ max 0 (cur!(i-1)+xsa!i) | i<-[0..length xs-1] ])
  best = array (-1,length xs-1) $ zip [-1..] (0:[ max (best!(i-1)) (cur!i) | i<-[0..length xs-1] ])

maxsubarray :: [Integer] -> [Integer]
maxsubarray xs = reverse $ snd $ maximum best
 where
  xsa = array (0,length xs-1) $ zip [0..] xs
  cur = array (-1,length xs-1) $ zip [-1..] ((0,[]):[ max (0,[]) (fst (cur!(i-1))+xsa!i,(xsa!i):(snd (cur
  best = array (-1,length xs-1) $ zip [-1..] ((0,[]):[ max (best!(i-1)) (cur!i) | i<-[0..length xs-1] ])
```

Complexité ?

Longest Increasing Subsequence

Étant donné une liste `xs`, trouver une sous–liste (pas forcément contigüe) croissante de taille maximum.
Exemples :

```
*Main> longest [1,5,2,9,3,8,4]
[1,2,3,4]
*Main> longest [5,4,3,2]
[2]
```

Longest Increasing Subsequence

Étant donné une liste `xs`, trouver une sous–liste (pas forcément contigüe) croissante de taille maximum.
Solution facile :

```
longest :: [Integer] -> [Integer]
longest [] = []
longest (x:xs) = if l1>l2 then sol1 else sol2
 where
  sol1 = x:(longest $ filter (>x) xs)
  sol2 = longest xs
  l1 = length sol1
  l2 = length sol2
```

Complexité ?

Longest Increasing Subsequence

Étant donné une liste `xs`, trouver une sous–liste (pas forcément contigüe) croissante de taille maximum.
Solution avec programmation dynamique :

```
longest :: [Integer] -> [Integer]
longest [] = []
longest xs = best lls
 where
  ll 0 = [head xs]
  ll i = (best [ lls !! j | j<-[0..i-1], xs!!j<xs!!i ])
           ++ [xs !! i]
  lls = [ ll i | i<-[0..length xs -1] ]

best :: [[Integer]] -> [Integer]
best sols = foldr
          (\x s->if length x>length s then x else s)
                                          [] sols
```

Complexité ?