# Informatique II (J1CP3020)

Philippe Duchon

U. Bordeaux 1

2012-13

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces
- Amène **naturellement** à des algorithmes récursifs

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces
- Amène **naturellement** à des algorithmes récursifs
- Idée : pour résoudre un problème de taille $n$ :

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces
- Amène **naturellement** à des algorithmes récursifs
- Idée : pour résoudre un problème de taille $n$ :
  - Si $n$ est “petit”, on le résoud directement

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces
- Amène **naturellement** à des algorithmes récursifs
- Idée : pour résoudre un problème de taille $n$ :
  - Si $n$ est “petit”, on le résoud directement
  - Sinon,

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces
- Amène **naturellement** à des algorithmes récursifs
- Idée : pour résoudre un problème de taille $n$ :
  - Si $n$ est “petit”, on le résoud directement
  - Sinon,
    - On le “découpe” en sous-problèmes plus petits (par exemple, 2 de taille $n/2$)

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces
- Amène **naturellement** à des algorithmes récursifs
- Idée : pour résoudre un problème de taille $n$ :
  - Si $n$ est “petit”, on le résoud directement
  - Sinon,
    - On le “découpe” en sous-problèmes plus petits (par exemple, 2 de taille $n/2$)
    - On résoud chaque sous-problème

# “Diviser pour régner”



- Une technique **très souvent employée** pour concevoir des algorithmes efficaces
- Amène **naturellement** à des algorithmes récursifs
- Idée : pour résoudre un problème de taille $n$ :
  - Si $n$ est “petit”, on le résoud directement
  - Sinon,
    - On le “découpe” en sous-problèmes plus petits (par exemple, 2 de taille $n/2$)
    - On résoud chaque sous-problème
    - On trouve la solution du “gros” problème en composant les solutions des sous-problèmes

# Un exemple (facile et artificiel)



- **Problème :** dans une liste $L$, trouver le maximum

# Un exemple (facile et artificiel)



- **Problème :** dans une liste $L$, trouver le maximum
- Si la liste $L$ ne comporte qu'un seul élément : renvoyer cet élément

# Un exemple (facile et artificiel)



- **Problème :** dans une liste $L$, trouver le maximum
- Si la liste $L$ ne comporte qu'un seul élément : renvoyer cet élément
- Sinon, couper la liste de $\ell$ éléments en deux listes $L'$ et $L''$ de $\lfloor \ell/2 \rfloor$ et $\lceil \ell/2 \rceil$ éléments

# Un exemple (facile et artificiel)



- **Problème :** dans une liste $L$, trouver le maximum
- Si la liste $L$ ne comporte qu'un seul élément : renvoyer cet élément
- Sinon, couper la liste de $\ell$ éléments en deux listes $L'$ et $L''$ de $\lfloor \ell/2 \rfloor$ et $\lceil \ell/2 \rceil$ éléments
- Trouver le maximum $x$ de $L'$ et $y$ de $L''$, et retourner $\max(x, y)$.

# Le maximum en Python



```
def Maxi(L):
    return MaxiRec(L,0,len(L))

def MaxiRec(L,g,d):
    if g==d:
        return(L[g])
    else:
        m=(g+d)/2
        x=MaxiRec(L,g,m)
        y=MaxiRec(L,m+1,d)
        if (x<y):
            z=y
        else:
            z=x
    return(z)
```

# Est-ce utile ?



- On avait un algorithme (non récursif) qui calculait le maximum en $n-1$ comparaisons ; est-ce qu'on fait mieux ?

# Est-ce utile ?



- On avait un algorithme (non récursif) qui calculait le maximum en $n-1$ comparaisons ; est-ce qu'on fait mieux ?
- On écrit une récurrence sur le nombre $C(n)$ de comparaisons faites pour trouver le maximum d'une liste de $n$ :

# Est-ce utile ?



- On avait un algorithme (non récursif) qui calculait le maximum en $n-1$ comparaisons ; est-ce qu'on fait mieux ?
- On écrit une récurrence sur le nombre $C(n)$ de comparaisons faites pour trouver le maximum d'une liste de $n$ :
  - $C(1) = 0$

# Est-ce utile ?



- On avait un algorithme (non récursif) qui calculait le maximum en $n-1$ comparaisons ; est-ce qu'on fait mieux ?
- On écrit une récurrence sur le nombre $C(n)$ de comparaisons faites pour trouver le maximum d'une liste de $n$ :
  - $C(1) = 0$
  - Pour $n \geq 2$, $C(n) = 1 + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil)$

# Est-ce utile ?



- On avait un algorithme (non récursif) qui calculait le maximum en $n-1$ comparaisons ; est-ce qu'on fait mieux ?
- On écrit une récurrence sur le nombre $C(n)$ de comparaisons faites pour trouver le maximum d'une liste de $n$ :
  - $C(1) = 0$
  - Pour $n \geq 2$, $C(n) = 1 + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil)$
- **Exercice :** Montrer qu'il existe une unique suite $(C(n))_{n \geq 1}$ qui satisfait cette récurrence. Montrer que, pour tout $n \geq 1$, $C(n) = n - 1$.

# Est-ce utile ?



- On avait un algorithme (non récursif) qui calculait le maximum en $n-1$ comparaisons ; est-ce qu'on fait mieux ?
- On écrit une récurrence sur le nombre $C(n)$ de comparaisons faites pour trouver le maximum d'une liste de $n$ :
  - $C(1) = 0$
  - Pour $n \geq 2$, $C(n) = 1 + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil)$
- **Exercice :** Montrer qu'il existe une unique suite $(C(n))_{n \geq 1}$ qui satisfait cette récurrence. Montrer que, pour tout $n \geq 1$, $C(n) = n - 1$.
- **Conclusion :** cet algorithme ne fait pas moins de comparaisons que l'algorithme itératif.

# Application au tri



- **Fusion** de deux listes triées : à partir de deux listes déjà triées dans l'ordre croissant, obtenir une liste triée, contenant l'union des éléments des deux listes de départ (algorithme non récursif)

# Application au tri



- **Fusion** de deux listes triées : à partir de deux listes déjà triées dans l'ordre croissant, obtenir une liste triée, contenant l'union des éléments des deux listes de départ (algorithme non récursif)
- Si on s'y prend bien, la fusion de 2 listes de longueurs $\ell_1$ et $\ell_2$, se fait en (au plus) $\ell_1 + \ell_2 - 1$ comparaisons de valeurs, et au moins $\min(\ell_1, \ell_2)$.

# Application au tri



- **Fusion** de deux listes triées : à partir de deux listes déjà triées dans l'ordre croissant, obtenir une liste triée, contenant l'union des éléments des deux listes de départ (algorithme non récursif)
- Si on s'y prend bien, la fusion de 2 listes de longueurs $\ell_1$ et $\ell_2$, se fait en (au plus) $\ell_1 + \ell_2 - 1$ comparaisons de valeurs, et au moins $\min(\ell_1, \ell_2)$.
- **Tri fusion :** en utilisant l'algorithme de fusion comme "recomposition", on écrit un algorithme "diviser pour régner" pour trier une liste.

# Application au tri



- **Fusion** de deux listes triées : à partir de deux listes déjà triées dans l'ordre croissant, obtenir une liste triée, contenant l'union des éléments des deux listes de départ (algorithme non récursif)
- Si on s'y prend bien, la fusion de 2 listes de longueurs $\ell_1$ et $\ell_2$, se fait en (au plus) $\ell_1 + \ell_2 - 1$ comparaisons de valeurs, et au moins $\min(\ell_1, \ell_2)$.
- **Tri fusion :** en utilisant l'algorithme de fusion comme "recomposition", on écrit un algorithme "diviser pour régner" pour trier une liste.
- **Récurrence** pour la complexité dans le cas le pire : $C(1) = 0$, et pour $n \geq 2$,

$$C(n) = n - 1 + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil)$$