Complexité algorithmique

Bruno Grenet

M1 MEEF Maths option Info – 2018 - 2019

Sommaire

Complexité d'un algorithme

Objectifs

- *Prédiction* du temps d'exécution
- *Comparaison* entre algorithmes

Méthodes

- Expérimentation : lancer le(s) algorithme(s) sur des entrées
- Analyse de complexité théorique

Exemples

- De combien de temps ai-je besoin pour calculer 1000! ?
- Recherche dichotomique ou recherche séquentielle ?

Complexité expérimentale

- Dépend des langages, machines, systèmes d'expoitation, ...
- Et donc varie au cours du temps !
- Mais utile et utilisée en pratique

Analyse théorique de la complexité

- S'applique aux *algorithmes*, écrits dans n'importe quel langage
- Résultat mathématique, donc fiable...
- ... si la modélisation est bonne !

Comment étudier la complexité d'un algorithme ?

Idée(s) 1 : Compter les opérations élémentaires

Raison : indépendant de la machine

- Une opération élémentaire = une unité de temps de calcul
- Additions, comparaisons, opérations sur les bits, . . .

  $\rightsquigarrow$ dépend du contexte
- En négligeant ce qui est négligeable

Idée(s) 1 : Compter les opérations élémentaires

Raison : indépendant de la machine

- Une opération élémentaire = une unité de temps de calcul
- Additions, comparaisons, opérations sur les bits, ...

  $\rightsquigarrow$ dépend du contexte
- En négligeant ce qui est négligeable

```
s = 0
for i in range(100):
    s += 1
```

Complexité

- 100 additions
- Affectations, gestion du compteur ignorées

Idée 2 : Complexité *en fonction* des entrées

Raison : comportement pour différentes tailles

Idée 2 : Complexité *en fonction* des entrées

Raison : comportement pour différentes tailles

```
def mul_naif(m, n):
    s = 0
    while m > 0:
        s += n
        m -= 1
    return s
```

Complexité

- m additions de nombres
- Coût d'une addition de nombres de k bits ?

Idée 3 : Complexité asymptotique

Raison : abstraction

- *Vrai* coût difficile à estimer, et dépendant de la machine

Idée 3 : Complexité asymptotique

Raison : abstraction

- *Vrai* coût difficile à estimer, et dépendant de la machine

Exemples

- Addition de deux nombres de k bits : $\mathcal{O}(k)$ opérations
- Évaluation d'un polynôme de degré d : $\mathcal{O}(d)$ opérations

```
def horner(P, x):
    d = len(P)-1
    s = P[d]
    for i in range(d-1, -1, -1): # d-1, d-2, ..., 0
        s = x * s + P[i]
    return s
```

Idée 4 : Pire cas

Raison : garantie de performance

Idée 4 : Pire cas

Raison : garantie de performance

Exemple : chercher un élément `x` dans un tableau `T`

```
i = 0
while i < len(T):
    if x == T[i]: return i
    i += 1
raise ValueError("{} n'apparaît pas dans le tableau"
        .format(x))
```

- Si `x` est le premier élément : 1 test
- Si `x` est le dernier élément : `len(T)` tests

$\rightsquigarrow$ Complexité $\mathcal{O}(\text{taille}(T))$

Retours sur les 4 idées

1. Compter le nombre d'opérations élémentaires...
2. ... en fonction des entrées ...
3. ... de manière asymptotique ...
4. ... dans le pire cas.

Retours sur les 4 idées

1. Compter le nombre d'opérations élémentaires...
2. ... en fonction des entrées ...
3. ... de manière asymptotique ...
4. ... dans le pire cas.

Discussion

1. Autres mesures : temps parallèle, espace mémoire, etc.
2. Taille en pratique des entrées ?
3. Constantes cachées, comportement pratique
4. Analyse en moyenne, analyse lissée (hors programme)

Parler d'asymptotique

Notations de Landau

Définition

Soit $f, g : \mathbb{N} \to \mathbb{R}_+$.

$f = \mathcal{O}(g)$	$\exists c, f(n) \leq cg(n)$	pour $n > n_0$
$f = o(g)$	$\forall c, f(n) \leq cg(n)$	pour $n > n_0$
$f = \Omega(g)$	$\exists c, f(n) \geq cg(n)$	pour $n > n_0$
$f = \Theta(g)$	$\exists c_1, c_2, c_1 g(n) \leq f(n) \leq c_2 g(n)$	

Notations de Landau

Définition

Soit $f, g : \mathbb{N} \to \mathbb{R}_+$.

$f = \mathcal{O}(g)$	$\exists c, f(n) \leq cg(n)$	pour $n > n_0$
$f = o(g)$	$\forall c, f(n) \leq cg(n)$	pour $n > n_0$
$f = \Omega(g)$	$\exists c, f(n) \geq cg(n)$	pour $n > n_0$
$f = \Theta(g)$	$\exists c_1, c_2, c_1 g(n) \leq f(n) \leq c_2 g(n)$	

Remarque (exercice)

- $f = \mathcal{O}(g) \iff g = \Omega(f)$
- $f = \Theta(g) \iff f = \mathcal{O}(g) \wedge f = \Omega(g)$
- $f = \mathcal{O}(g)$ et $g = \mathcal{O}(h) \implies f = \mathcal{O}(h)$
- $f = \Omega(g)$ et $g = \Omega(h) \implies f = \Omega(h)$
- $f = \Theta(g)$ et $g = \Theta(h) \implies f = \Theta(h)$

Les classiques

- Logarithmique : $\mathcal{O}(\log n)$ (polylog. : $\mathcal{O}(\log^k n)$ pour $k \geq 1$)
- Linéaire : $\mathcal{O}(n)$ (quasilinéaire : $\mathcal{O}(n \log^k n)$)
- Quadratique : $\mathcal{O}(n^2)$
- Cubique : $\mathcal{O}(n^3)$
- Polynomial : $\mathcal{O}(n^k)$ pour un certain $k \geq 1$
- Exponentiel : $\mathcal{O}(2^n)$ ou $\mathcal{O}(k^n)$ pour $k > 1$

Les classiques

- Logarithmique : $\mathcal{O}(\log n)$ (polylog. : $\mathcal{O}(\log^k n)$ pour $k \geq 1$)
- Linéaire : $\mathcal{O}(n)$ (quasilinéaire : $\mathcal{O}(n \log^k n)$)
- Quadratique : $\mathcal{O}(n^2)$
- Cubique : $\mathcal{O}(n^3)$
- Polynomial : $\mathcal{O}(n^k)$ pour un certain $k \geq 1$
- Exponentiel : $\mathcal{O}(2^n)$ ou $\mathcal{O}(k^n)$ pour $k > 1$

Efficace ?

- *Efficace* $\simeq$ polynomial
- Discutable :
  - $\mathcal{O}(1.000001^n)$ vs $\mathcal{O}(n^{10000})$
  - $\mathcal{O}(n^2)$ pour $n = 1\,000\,000$

Les classiques

- Logarithmique : $\mathcal{O}(\log n)$ (polylog. : $\mathcal{O}(\log^k n)$ pour $k \geq 1$)
- Linéaire : $\mathcal{O}(n)$ (quasilinéaire : $\mathcal{O}(n \log^k n)$)
- Quadratique : $\mathcal{O}(n^2)$
- Cubique : $\mathcal{O}(n^3)$
- Polynomial : $\mathcal{O}(n^k)$ pour un certain $k \geq 1$
- Exponentiel : $\mathcal{O}(2^n)$ ou $\mathcal{O}(k^n)$ pour $k > 1$

Efficace ?

- *Efficace* $\simeq$ polynomial
- Discutable :
  - $\mathcal{O}(1.000001^n)$ vs $\mathcal{O}(n^{10000})$
  - $\mathcal{O}(n^2)$ pour $n = 1\,000\,000$

Attention à la taille de l'entrée !

Exemples

```
def min(T):
    m = T[0]
    for t in T[1:]:
        if t < m: m = t
    return m


def mul_naif(m,n)
    s = 0
    while n > 0:
        s += m
        n -= 1
    return s
```

Exemples

```
def min(T):
    m = T[0]
    for t in T[1:]:
        if t < m: m = t
    return m
```

```
def mul_naif(m,n)
    s = 0
    while n > 0:
        s += m
        n -= 1
    return s
```

Complexités

- Dans les deux cas $\mathcal{O}(n)$
- Tailles d'entrée :
  - `min`: $n \times \text{taille}(t)$
  - `mul_naif`: $\log mn$

Complexité des algorithmes récursifs

```
def Fact(n):
    if n < 2: return 1
    return n * Fact(n-1)
```

Complexité

- Soit $t(n)$ le nombre de multiplications faites par `Fact(n)`.
- Alors $t(n) = 1 + t(n-1)$ pour $n > 2$.
- On résout la récurrence : $t(n) = n - 1$ (exercice).
- Complexité asymptotique : $\mathcal{O}(n)$ multiplications
- Aller plus loin : taille des entiers ?

Complexité en espace

- Compter l'espace mémoire utilisé au cours du calcul
- Taille de stockage des entrées ignoré $\rightsquigarrow$ plus de finesse
- Souvent moins crucial que la complexité en temps...
- ... mais explique certaines déconvenues !

Complexité en espace

- Compter l'espace mémoire utilisé au cours du calcul
- Taille de stockage des entrées ignoré $\rightsquigarrow$ plus de finesse
- Souvent moins crucial que la complexité en temps...
- ... mais explique certaines déconvenues !

Ordres de grandeurs différents

- *Efficace* : $\mathcal{O}(\log^k n)$ pour un certain k
- Moyen : $\mathcal{O}(n)$
- Inefficace : $\mathcal{O}(n^k)$, $k > 1$

Exemple : $F_n = F_{n-1} + F_{n-2}$

```python
def Fibo1(n):
    if n < 2: return 1
    return Fibo1(n-1) + Fibo1(n-2)

def Fibo2(n):
    if n < 2: return 1
    T = [1] * (n+1)
    for i in range(2, n+1):
        T[i] = T[i-1] + T[i-2]
    return T[n]
```

Complexités

- $t_1(n) \leq t_1(n-1) + t_1(n-2) \rightsquigarrow t_1(n) = \mathcal{O}(2^n)$
- $t_2(n) \leq \mathcal{O}(n)$ mais espace mémoire nécessaire $\mathcal{O}(n)$

Version améliorée

```
def Fibo3(n):
    if n < 2: return 1
    F1 = F2 = 1
    for i in range(2, n+1):
        F1, F2 = F1 + F2, F1
    return F1
```

Complexité

- Toujours $\mathcal{O}(n)$
- Espace mémoire : deux variables entières

Version améliorée

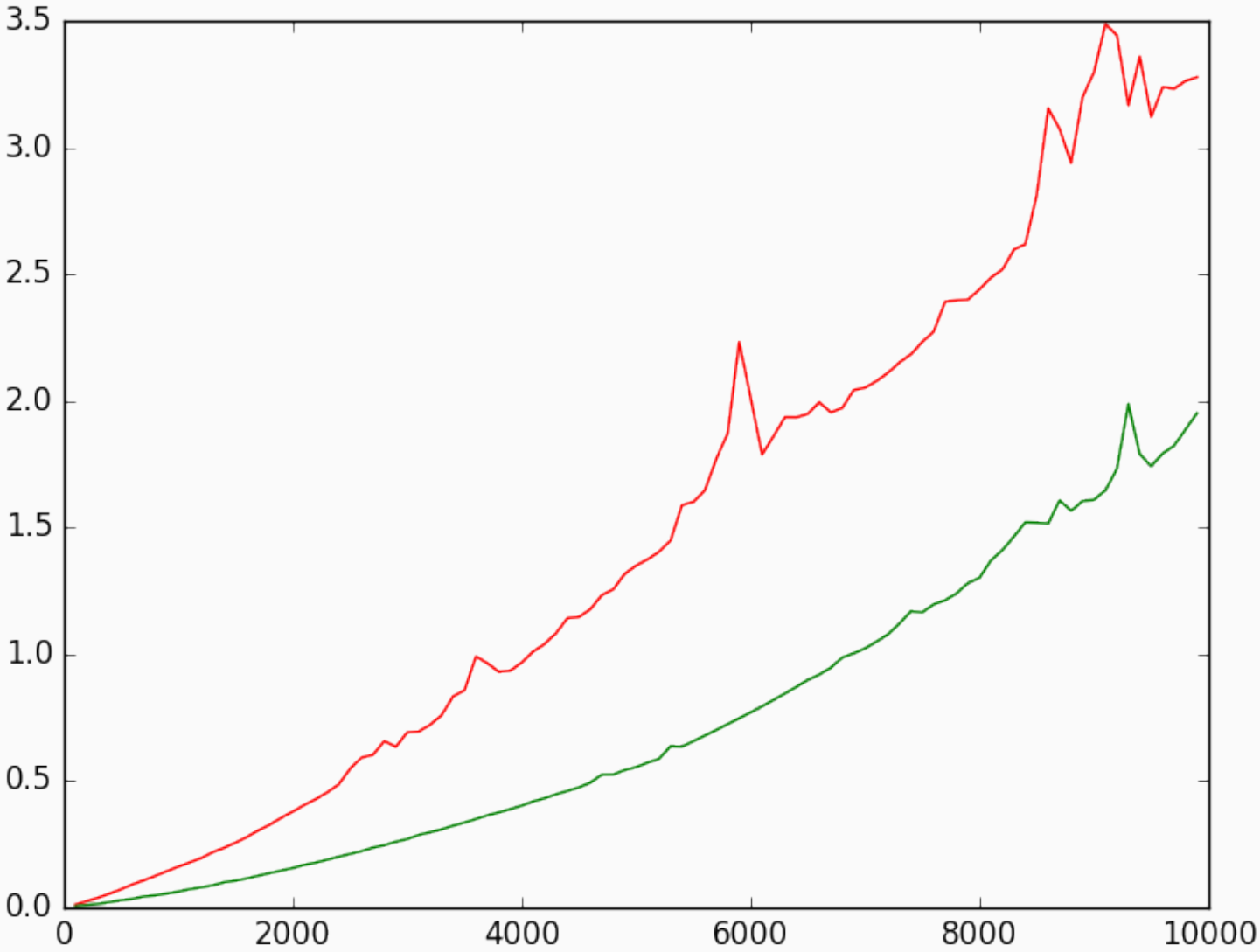
```
def Fibo3(n):
    if n < 2: return 1
    F1 = F2 = 1
    for i in range(2, n+1):
        F1, F2 = F1 + F2, F1
    return F1
```

Complexité

- Toujours $\mathcal{O}(n)$
- Espace mémoire : deux variables entières

Conclusion
Le n-ième élément de la suite de Fibonacci se calcule en temps $\mathcal{O}(n)$, avec un espace $\mathcal{O}(\log n)$.

Influence de la complexité en espace



Techniques algorithmiques

Calcul de x^n

- Entrée : x et n où n est un entier positif
- Sortie : x^n
- Modèle : on ne suppose rien sur x, à part qu'on peut le multiplier (monoïde multiplicatif)
- But : trouver un ou des algorithmes pour cela, et analyser leur complexité

Algorithme naïf

```
def puiss_naif(x, n):
    r = x
    for i in range(n-1):
        r *= x
    return r
```

Complexité. $\mathcal{O}(n)$

Méthode binaire

On remarque

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{si } n \text{ est pair,} \\ x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x & \text{sinon.} \end{cases}$$

Méthode binaire

On remarque

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{si } n \text{ est pair,} \\ x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x & \text{sinon.} \end{cases}$$

```
def puiss_bin(x, n):
    if n == 1: return x
    tmp = puiss_bin(x, n//2)
    if n % 2:
        return tmp * tmp * x
    return tmp * tmp
```

Complexité. $t(n) \leq t(\lfloor n/2 \rfloor) + 2 \implies t(n) \leq \mathcal{O}(\log n)$

Méthode des facteurs

On remarque

$$x^n = \begin{cases} (x^p)^{(n/p)} & \text{si } p \text{ divise } n, \\ x^{n-1} \cdot x & \text{si } p \text{ est premier.} \end{cases}$$

Méthode des facteurs

On remarque

$$x^n = \begin{cases} (x^p)^{(n/p)} & \text{si } p \text{ divise } n, \\ x^{n-1} \cdot x & \text{si } p \text{ est premier.} \end{cases}$$

```
def puiss_facteurs(x, n):
    if n == 1: return x
    p = plus_petit_facteur_premier(n)
    if p == n:
        return puiss_facteurs(x, n-1) * x
    return puiss_facteurs(puiss_facteurs(x, p), n//p)
```

Complexité. ???

Techniques algorithmiques

Diviser pour régner

Produit de polynômes

- Entrée : P, Q deux polynômes, représentés par le tableau de leurs n coefficients
- Sortie : $R = PQ$, représenté par le tableau de ses coefficients

Produit de polynômes

- Entrée : P, Q deux polynômes, représentés par le tableau de leurs n coefficients
- Sortie : $R = PQ$, représenté par le tableau de ses coefficients

```
def prod_naif(P, Q):
    R = [0] * (len(P) + len(Q) - 1)
    for i in range(len(R)):
        for j in range(i+1):
            R[i] += P[j] * Q[i-j]
    return R
```

Complexité. $\mathcal{O}(n^2)$ opérations

Méthode de Karatsuba

Soit $m = \lfloor n/2 \rfloor$. On peut écrire $P = P_1 + x^m P_2$ et $Q = Q_1 + x^m Q_2$, avec P_1, P_2, Q_1, Q_2 de degré $\leq m$. Alors

$$R = P_1 Q_1 + (P_1 Q_2 + P_2 Q_1) x^m + P_2 Q_2 x^{2m}.$$

Méthode de Karatsuba

Soit $m = \lfloor n/2 \rfloor$. On peut écrire $P = P_1 + x^m P_2$ et $Q = Q_1 + x^m Q_2$, avec P_1, P_2, Q_1, Q_2 de degré $\leq m$. Alors

$$R = P_1 Q_1 + (P_1 Q_2 + P_2 Q_1) x^m + P_2 Q_2 x^{2m}.$$

On remarque

$$P_1 Q_2 + P_2 Q_1 = (P_1 + P_2) \cdot (Q_1 + Q_2) - P_1 Q_1 - P_2 Q_2.$$

Méthode de Karatsuba

Soit $m = \lfloor n/2 \rfloor$. On peut écrire $P = P_1 + x^m P_2$ et $Q = Q_1 + x^m Q_2$, avec P_1, P_2, Q_1, Q_2 de degré $\leq m$. Alors

$$R = P_1 Q_1 + (P_1 Q_2 + P_2 Q_1) x^m + P_2 Q_2 x^{2m}.$$

On remarque

$$P_1 Q_2 + P_2 Q_1 = (P_1 + P_2) \cdot (Q_1 + Q_2) - P_1 Q_1 - P_2 Q_2.$$

Algorithme

1. Couper P en $P_1 + x^m P_2$ et Q en $Q_1 + x^m Q_2$;
2. Calculer $R_1 = P_1 Q_1$, $R_2 = P_2 Q_2$, $R_3 = (P_1 + P_2)(Q_1 + Q_2)$
3. Renvoyer $R_1 + (R_3 - R_2 - R_1) x^m + R_2 x^{2m}$

Analyse de complexité

Soit $T(n)$ le nombre d'additions et multiplications de coefficients pour deux polynômes de degrés $n-1$

- Supposons que $n = 2^k$. Alors $T(2^k) \leq 3T(2^{k-1}) + \mathcal{O}(2^k)$

Analyse de complexité

Soit $T(n)$ le nombre d'additions et multiplications de coefficients pour deux polynômes de degrés $n-1$

- Supposons que $n = 2^k$. Alors $T(2^k) \leq 3T(2^{k-1}) + \mathcal{O}(2^k)$
- Notons $T_k \leq 3T_{k-1} + s_k$ avec $s_k \leq c2^k$:

$$T_k \leq 3T_{k-1} + s_k \leq 3(3T_{k-2} + s_{k-1}) + s_k \leq 3^k T_0 + \sum_{j<k} 3^j s_{k-j}$$

$$\sum_{j<k} 3^j s_{k-j} \leq c2^k \sum_{j<k} (3/2)^j = c2^k \frac{(3/2)^k - 1}{3/2 - 1} = 2c(3^k - 2^k)$$

Analyse de complexité

Soit $T(n)$ le nombre d'additions et multiplications de coefficients pour deux polynômes de degrés $n-1$

- Supposons que $n = 2^k$. Alors $T(2^k) \leq 3T(2^{k-1}) + \mathcal{O}(2^k)$
- Notons $T_k \leq 3T_{k-1} + s_k$ avec $s_k \leq c2^k$:

$$T_k \leq 3T_{k-1} + s_k \leq 3(3T_{k-2} + s_{k-1}) + s_k \leq 3^k T_0 + \sum_{j<k} 3^j s_{k-j}$$

$$\sum_{j<k} 3^j s_{k-j} \leq c2^k \sum_{j<k} (3/2)^j = c2^k \frac{(3/2)^k - 1}{3/2 - 1} = 2c(3^k - 2^k)$$

- Donc $T_k = \mathcal{O}(3^k) = \mathcal{O}(2^{k \log 3}) = \mathcal{O}(n^{\log 3})$

Analyse de complexité

Soit $T(n)$ le nombre d'additions et multiplications de coefficients pour deux polynômes de degrés $n-1$

- Supposons que $n = 2^k$. Alors $T(2^k) \leq 3T(2^{k-1}) + \mathcal{O}(2^k)$
- Notons $T_k \leq 3T_{k-1} + s_k$ avec $s_k \leq c2^k$:

$$T_k \leq 3T_{k-1} + s_k \leq 3(3T_{k-2} + s_{k-1}) + s_k \leq 3^k T_0 + \sum_{j<k} 3^j s_{k-j}$$

$$\sum_{j<k} 3^j s_{k-j} \leq c2^k \sum_{j<k} (3/2)^j = c2^k \frac{(3/2)^k - 1}{3/2 - 1} = 2c(3^k - 2^k)$$

- Donc $T_k = \mathcal{O}(3^k) = \mathcal{O}(2^{k \log 3}) = \mathcal{O}(n^{\log 3})$
- Pour tout n, il y a une puissance 2^k inférieure à $2n$

Analyse de complexité

Soit $T(n)$ le nombre d'additions et multiplications de coefficients pour deux polynômes de degrés $n-1$

- Supposons que $n = 2^k$. Alors $T(2^k) \leq 3T(2^{k-1}) + \mathcal{O}(2^k)$
- Notons $T_k \leq 3T_{k-1} + s_k$ avec $s_k \leq c2^k$:

$$T_k \leq 3T_{k-1} + s_k \leq 3(3T_{k-2} + s_{k-1}) + s_k \leq 3^k T_0 + \sum_{j<k} 3^j s_{k-j}$$

$$\sum_{j<k} 3^j s_{k-j} \leq c2^k \sum_{j<k} (3/2)^j = c2^k \frac{(3/2)^k - 1}{3/2 - 1} = 2c(3^k - 2^k)$$

- Donc $T_k = \mathcal{O}(3^k) = \mathcal{O}(2^{k \log 3}) = \mathcal{O}(n^{\log 3})$
- Pour tout n, il y a une puissance 2^k inférieure à $2n$
- Donc $T(n) \leq T(2^k) \leq \mathcal{O}((2n)^{\log 3}) = \mathcal{O}(n^{\log 3})$

Techniques algorithmiques

Algorithmes gloutons

Utilisation d'un gymnase

- Entrée : un ensemble A d'activités $A_i = (d_i, f_i)$ [début, fin]
- Sortie : une suite ordonnée de $(A_{i_1}, \ldots, A_{i_k})$ telle que
  - pour tout $j < k$, $f_{i_j} \leq d_{i_{j+1}}$
  - k est maximal : il n'existe pas de suite de taille $k+1$ satisfaisant la première condition

Utilisation d'un gymnase

- Entrée : un ensemble A d'activités $A_i = (d_i, f_i)$ [début, fin]
- Sortie : une suite ordonnée de $(A_{i_1}, \ldots, A_{i_k})$ telle que
  - pour tout $j < k$, $f_{i_j} \leq d_{i_{j+1}}$
  - k est maximal : il n'existe pas de suite de taille $k+1$ satisfaisant la première condition

Algorithme glouton
À chaque étape, on choisit un *minimum local*.

Algorithme générique

```
def glouton(A, critere):
    A.sort(key = critere)
    S = [A[0]]
    for (d,f) in A[1:]:
        i = 0
        while i < len(S) and S[i][1] <= d: i += 1
        if i == len(S): S.append((d,f))
        elif S[i][0] >= f: S.insert(i, (d,f))
        if d > S[-1][1]: S.append((d,f))
    return S
```

Algorithme générique

```
def glouton(A, critere):
    A.sort(key = critere)
    S = [A[0]]
    for (d,f) in A[1:]:
        i = 0
        while i < len(S) and S[i][1] <= d: i += 1
        if i == len(S): S.append((d,f))
        elif S[i][0] >= f: S.insert(i, (d,f))
        if d > S[-1][1]: S.append((d,f))
    return S
```

- Durées croissantes : `def critere(a): return a[1]-a[0]`
- Dates de début croissantes : `def critere(a): return a[0]`
- Dates de fin croissantes : `def critere(a): return a[1]`

Quel choix de critère ?

- Durées croissantes : $A = [(1, 5), (4, 7), (6, 10)]$

Quel choix de critère ?

- Durées croissantes : $A = [(1, 5), (4, 7), (6, 10)]$
- Dates de début croissantes :
  $A = [(1, 10), (2, 3), (4, 5), (6, 7), (8, 9)]$

Quel choix de critère ?

- Durées croissantes : $A = [(1, 5), (4, 7), (6, 10)]$
- Dates de début croissantes :
  $A = [(1, 10), (2, 3), (4, 5), (6, 7), (8, 9)]$
- Dates de fin croissantes : optimal !

Quel choix de critère ?

- Durées croissantes : $A = [(1, 5), (4, 7), (6, 10)]$
- Dates de début croissantes :
  $A = [(1, 10), (2, 3), (4, 5), (6, 7), (8, 9)]$
- Dates de fin croissantes : optimal !

Quel choix de critère ?

- Durées croissantes : $A = [(1,5),(4,7),(6,10)]$
- Dates de début croissantes :
  $A = [(1,10),(2,3),(4,5),(6,7),(8,9)]$
- Dates de fin croissantes : optimal !

Preuve. Récurrence sur $n = |A|$. Si $n = 1$, résultat trivial.

- L'algo. choisit A_1.
- Ensuite, l'algo. calcule une solution S' pour l'entrée
  $A' = \{A_j : d_j > f_1\}$.
- Par hyp. de réc., S' est optimale pour A'.
- On montre que $\{A_1\} \cup S'$ est optimale pour A :
  - Soit S une solution (optimale)
  - S contient au plus 1 élément de $A \setminus A' = \{A_j : d_j \leq f_1\}$
  - Donc $S = \{A_{i_1}\} \cup S''$ avec S'' sol. de $A'' = \{A_j : d_j > f_{i_1}\} \subset A'$

Techniques algorithmiques

Programmation dynamique

Rendu de monnaie

- Entrée : une somme S et une liste C de valeurs de pièces
- Sortie : une liste de pièces dont la somme vaut S

```
def pieces_glouton(S, C):
    reste = S
    pieces = {c:0 for c in C}
    while reste > 0:
        cmax = max(c for c in C if c <= reste)
        reste -= cmax
        pieces[cmax] += 1
    return pieces
```

Non optimal
Pièces $(4, 3, 1)$ et somme 6

Programmation dynamique pour le rendu de monnaie

Remarque
$\#\text{pieces}_C(S) = 1 + \min_{c \in C}(\#\text{pieces}_C(S - c))$

```
def pieces_prog_dyn(S, C):
    T = [0]
    for i in range(1, S+1):
        T.append(1 + min(T[i-c] for c in C if i-c >= 0))
    return T[S]
```

Optimal
Preuve : remarque ci-dessus !

Techniques algorithmiques

Conclusion

Comparaisons

- Glouton : choix *local* pour construire un minimum global
  - Complexité souvent linéaire
  - Difficulté : résultat pas toujours optimal
- Diviser-pour-régner : découpe en sous-problèmes indépendants
  - Complexité : équation de récurrence à résoudre
  - Attention à ne pas faire le même calcul plusieurs fois !
- Programmation dynamique : construction de la solution à partir de solutions partielles
  - Complexité proportionnelle au nombre de problèmes partiels
  - Mais l'espace aussi ! Peut-on le réduire ?

CAPES
Notions pas explicitement au programme, mais des algorithmes de chaque type peuvent apparaître

Classes P et NP

Classes de complexité

Idée. Regrouper les problèmes par *classes* :

- Une classe regroupe les problèmes ayant une *même borne supérieure de difficulté*
- Un outil de comparaison des problèmes

Exemples

- Les problèmes (au plus) linéaires: $\mathcal{O}(n)$
- Les problèmes (au plus) quadratiques: $\mathcal{O}(n^2)$
- Les problèmes (au plus) exponentiels: $\mathcal{O}(2^n)$

Remarque. « linéaire $\subset$ quadratique $\subset$ exponentiel »

Les problèmes de décision

Définition
Un *problème de décision* est un problème dont la sortie est binaire (0/1, `true`/`false`, oui/non, ...).

Exemples

- Le tableau est-il trié ?
- Est-ce que $m \times n = k$?

Remarque. On peut « toujours » transformer un problème en problème de décision équivalent :

- Minimum d'un tableau : « $min(T) = k$? », « $min(T) \leq k$? »
- Fibonacci : « $F_n = k$? »
- Tri : « $T[k]$ est le p-ième plus petit élément de T ? »

La classe P

Définition
La classe P est la classe des problèmes *de décision* de complexité polynomiale, c'est-à-dire de complexité $\mathcal{O}(n^k)$ pour un entier k.

Attention !
Dans la définition, n doit être la *taille de la représentation* de l'entrée en binaire : si l'entrée est un entier N, sa taille est $n = \lceil \log N \rceil$.

Rappel. P est la classe des problèmes considérés comme *faciles*.

Abus de langage. On dit parfois qu'un problème est dans P pour dire qu'il est de complexité polynomiale, même si ce n'est pas un problème de décision.

Exemples de problèmes dans P

Exemples

- Addition, multiplication, etc. d'entiers, de polynômes, etc.
- Recherche d'un élément dans un tableau, tri d'un tableau, ...
- Certains problèmes sur les graphes
- ...

La classe NP

Définition
La classe NP est la classe des problèmes de décision dont on peut *vérifier en temps polynomial* une solution donnée.

Exemples

- Circuit hamiltonien dans un graphe
- Somme d'un sous-ensemble :
  étant donné $S \subset \mathbb{Z}$, $\exists ? X \subset S$ tq $\sum_{x \in X} x = 0$
- Factorisation d'entiers
- Tout problème de la classe P ($\mathrm{P} \subset \mathrm{NP}$)

Appartenance à NP

Question
Comment montrer qu'un problème appartient à NP ?

Deux ingrédients

- Solution (*certificat*) de taille polynomiale
- Algorithme polynomial de vérification d'une solution

Exemple
Étant donné une grille de Sudoku en partie remplie, y a-t-il une solution ?

- Certificat : la solution complète
- Algorithme : vérification de chaque condition

Le problème « P = NP ? »

Question ouverte
Est-ce que P = NP ou P $\subsetneq$ NP ? Autrement dit, existe-t-il des problèmes dans NP qui ne sont pas dans P ?

Est-il plus difficile de trouver une solution que la vérifier ?

Exemples

- Peut-on calculer un circuit hamiltonien en temps polynomial ?
- Somme d'un sous-ensemble : peut-on déterminer s'il existe $X \subset S$ tq $\sum_{x \in X} x = 0$ en temps poly($|S|$) ?
- Peut-on factoriser un nombre n en temps poly(n) ?

Problèmes difficiles