Complexité des algorithmes

Stéphane Gonnord

stephane@gonnord.org
www.mp933.fr

Lycée du parc - Lyon

Vendredi 13 et 20 décembre 2013
Lycée du parc - PCSI 841

Plan

1. Ordres de grandeur
2. Un catalogue
   - Les calculs de tous les jours ;
   - des recherches ;
   - des tris ;
   - des calculs approchés.
3. Méthodes génériques
   - Une boucles...
   - puis deux.
   - mais aussi...
4. Dans le détail
   - Déjà vu dans les épisodes précédents.
   - Un tri.
   - L'algorithme d'Euclide.

Ordres de grandeur

Rappel : $10^9 = 2^{30} =$ une minute $=$ une seconde !

N	10^2	10^4	10^6	10^8
$\sqrt{N}$	10	10^2	10^3	10^4
$N\sqrt{N}$	10^3	10^6	10^9	10^{12}
$N\ln(N) \simeq$	$4,6 \times 10^2$	$9,2 \times 10^4$	$1,4 \times 10^7$	$1,8 \times 10^9$
N^2	10^4	10^8	10^{12}	10^{16}
N^3	10^6	10^{12}	10^{18}	10^{24}

Les calculs de tous les jours

- Addition sur p décimales/bits : $O(p)$.
- Multiplications :
  - $O(p^2)$...
  - ou $O(p^\alpha)$ avec $\alpha = \frac{\ln 3}{\ln 2} \simeq 1,6$...
  - voire $O(p \ln p)$.
- Division euclidienne : idem.
- Calcul de x^n : $n - 1 = n$ multiplications... ou $\ln n$.
- calcul de pgcd, pour p décimales/bits : $O(\ln^3 p)$ opérations élémentaires... et même $O(\ln^2 p)$ en finassant.
- Calcul matriciel : tout en n^3 (pivot) !
- Primalité : $\sqrt{n}(\ln n)^2$... puis $(\ln n)^2$ en version probabiliste... puis déterministe en $(\ln n)^{12}$ (2002 ; probablement $(\ln n)^6$).

Diverses recherches

- Recherche d'un maximum/minimum :
  - $n-1=n$ comparaisons pour l'algorithe naïf...
  - Et combien pour un algorithme «Roland Garros» ? $n-1$ aussi !
  - On peut faire mieux ? Ben non ! Mais ce n'est pas si clair...

- Appartenance à...
  - un tableau de taille n : n ou $\ln n$;
  - d'autres structures de données (arbre, table de hash...) : $\ln n$ si possible !

- Un mot m_1 dans un mot m_2 :
  - $|m_1| \times |m_2|$ en naïf...
  - mais $|m_1| + |m_2|$ en finassant (KMP).

Algorithme de tri

- Les basiques (sélection - dont bulles - ou insertion).

  Exécution en temps n^2

- Un peu mieux : tri fusion (dichotomique) et rapide

  Exécution en temps $n.\ln(n)$ (moyenne)

  donc utilisable pour $n = 10^6$*. Mais pire des cas en* n^2 *pour le tri rapide.*

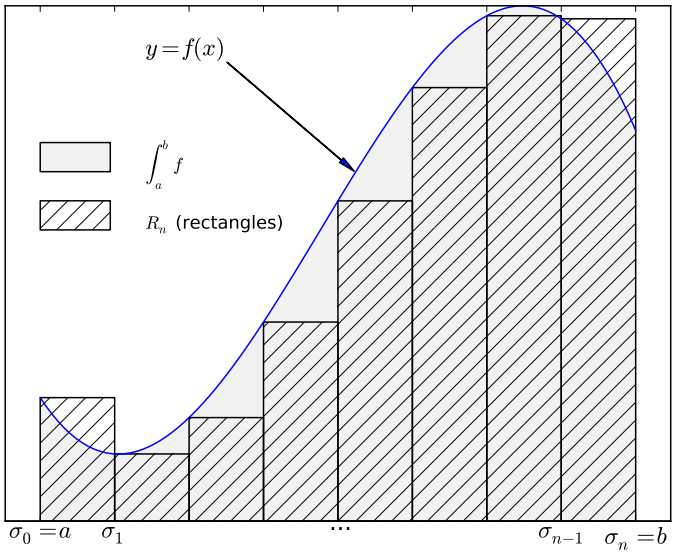
- Plus exotiques : tri par tas, tri shell, ...

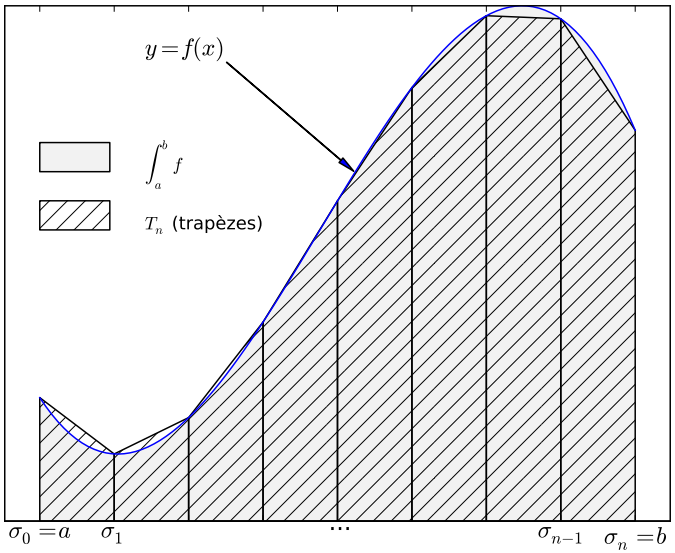
  Exécution en temps $n.\ln(n)$... ou $n.p$ sur p décimales

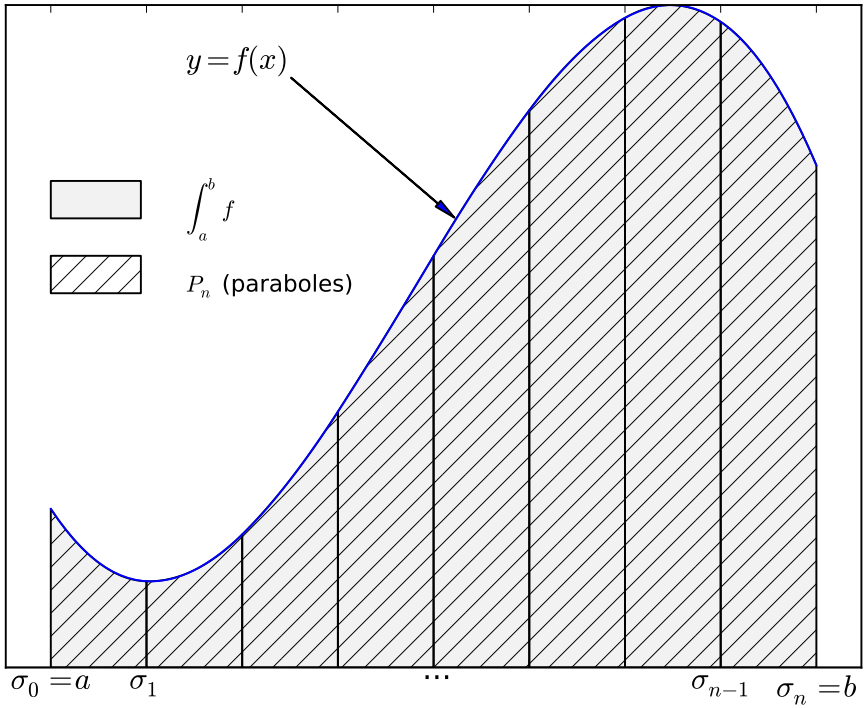
Calculs approchés

- Résolution numérique de $f(x) = 0$:
  - n décimales en $O(n)$ évaluations (dichotomie) : bien !
  - n décimales en $O(\ln n)$ évaluations (Newton, sécante) : encore mieux !
- Calcul approché d'une intégrale :
  - Avec n évaluations, approximation à $\frac{K}{n}$ avec des rectangles...
  - mais $\frac{K}{n^2}$ avec des trapèzes...
  - et même $\frac{K}{n^4}$ avec des paraboles.
- Résolution approchée d'une équation différentielle :
  - Approximation en $\frac{1}{n}$ avec n itérations (Euler)...
  - ou $\frac{1}{n^2}$ (Heun)...
  - ou même $\frac{1}{n^4}$ (RK4).

HS - juste pour frimer







Une boucle

- Situation typique :

```
for i in <un range, ou un ensemble, une liste>:
    <faire des
    trucs...>
```

- Complexité : autant de trucs qu'il y en a à faire !
- Exemples :
  - ```
    for i in range(10):
        print("Allo, non mais allo numéro "+str(i))
    ```
  - ```
    u = 0
    for _ in range(100):
        u = cos(u)
    ```
  - ```
    maxi = t[0]
    for x in t:
        if x > maxi:
            maxi = t
    ```
    ```

Cas des sorties prématurées

- `break` (sortie de boucle ; alternative au return/while)

```
def appartient(x, t):
    est_present = False
    for y in t:
        print(y)
        if y == x:
            est_present = True
            break
    print("paf")
    return est_present
```

- Ce qui donne :

```
>>> appartient(42, [12, 42, 1515])
12
42
paf
True
```

- Pire des cas, meilleur des cas, cas moyen...

Deux boucles

- Cas générique :

```
for x in foo:
    for y in bar:
        <faire_tel_truc>
```

- Si `foo` et `bar` sont constant : $|foo| \times |bar|$ exécutions de `tel_truc`
- Cas typique ou `bar` dépend de `foo` :

```
for i in range(1, 1+n):
    for j in range(i):
        <whatever>
```

- Complexité : $1 + 2 + \cdots + n$ exécutions de `whatever`, soit $\frac{n(n+1)}{2} = \frac{n^2}{2} = n^2$! Terme non standard parfois rencontré : « *complexité triangulaire* ».

Exercice

Évaluer le nombre d'exécutions de `pif`, `paf` et `pouf` lors de l'exécution des trois programmes suivants :

- ```
  for i in range(n):
      pif
  for j in range(n):
      paf
  for k in range(n):
      pouf
  ```
- ```
  for i in range(n):
      pif
      for j in range(n):
          paf
  for k in range(n):
      pouf
  ```
- ```
  for i in range(n):
      pif
      for j in range(n):
          paf
          for k in range(n):
              pouf
  ```

Hum... bon exercice de DS, vous ne trouvez pas ?
  ```

Les boucles while

- Cas générique :

```
while <condition qui porte sur un truc>:
    faire des machins
```

- Si les machins ne touchent pas aux trucs, c'est (probablement) la catastrophe... sauf en cas de sortie via un `break`
- Complexité et terminaison fortement liées.
- Exemples de couples conditions/actions :
  - `while n>0` `n = n-1`
  - `while n>0` `n = n//2`
  - `while n<M` `n = n*2`
  - `while g<d` `g++` ou `d--` ou `d-g` est divisé par 2
  - `while n>0` `n = n//2` ou `n = 3*n+1`

Algorithmes déjà rencontrés

- Calcul de x^n.
  $r \leftarrow x$
  pour *i allant de* 1 *à n* **faire**
  $\quad r \leftarrow r \times x$
  Résultat : r

- Calcul de $n!$ en n.
  $r \leftarrow 1$
  pour *i allant de* 2 *à n* **faire**
  $\quad r \leftarrow r \times i$
  Résultat : r

- Déterminer le plus petit entier n tel que $n! \geq 10^{10}$
  $n \leftarrow 1$
  $f \leftarrow 1$ # f vaudra $n!$ à (presque) tout moment
  tant que $f < 10^{10}$ **faire**
  $\quad n \leftarrow n+1$
  $\quad f \leftarrow f \times n$
  Résultat : n

Algorithmes/programmes déjà rencontrés

- Addition de deux entiers (tableaux de décimales)

```
def somme(t1, t2): # on les suppose de même longueur
    resultat, retenue = [], 0
    for i in range(len(t1)):
        somme = t1[i] + t2[i] + retenue
        resultat.append(somme % 10)
        retenue = somme // 10
    return resultat
```

  « C'est linéaire »

- Multiplication de deux entiers :
  - faire le produit de t_1 par chaque élément de t_2 ;
  - additionner les listes obtenues (avec décalages, et éventuellement à la volée).
  - Il y a $|t_1| \times |t_2|$ multiplications/additions.

Algorithmes/programmes déjà rencontrés

- Calcul de 2-valuation

```
cpt = 0
while n % 2 == 0:
    cpt = cpt+1
    n = n // 2
```

- Calcul de x^n (exponentiation rapide itérative)

```
def expo_rapide(x, n0):
    r, p, n = 1, x, n0
    while n > 0:
        if n%2 == 1:
            r = r * p
        p = p * p
        n = n // 2
    return r
```

Algorithmes/programmes déjà rencontrés

- Appartenance à un tableau

```
def appartient(x, t):
    for y in t:
        if y == x:
            return True
    return False
```

- Positions

```
def positions(x, t):
    pos = []
    for i in range(len(t)):
        if t[i] == x:
            pos.append(i)
    return pos
```

- Syracuse

```
while n>1:
    if n%2 ==0:
        n = n/2
    else:
        n = 3*n+1
```

Algorithmes/programmes déjà rencontrés

- Dichotomie

```
def recherche_dichotomique(b, t):
    d, f = 0, len(t)-1
    while f > d:
        m = (d+f+1)//2
        if t[m] == b:
            return True
        if t[m] >b:
            f = m-1
        else:
            d = m+1
    if d == f and t[d] == b:
        return True
    return False
```

- Crible

```
def crible(n):
    t = [False, False]+([True] * (n-1))
    for k in range(2, 1+int(sqrt(n))):
        for i in range(k, 1+(n//k)):
            t[k*i] = False
    return t
```

Un tri par sélection

- Algorithme principal :

  Entrées : T

  pour j *allant de* $n-1$ *à* 1 **faire**

  Trouver un indice i_0 tel que `T[i0]` soit maximal dans `T[0:j+1]` # donc jusqu'à j inclus...

  Échanger `T[i0]` et `T[j]`

  Résultat : T

- Algorithme secondaire (trouver l'indice...)

  Entrées : T, j

  $p \leftarrow 0$ # la position du maximum provisoire **pour** i *allant de* 1 *à* j **faire**

  si $T[i] > T[p]$ **alors**

  $p \leftarrow i$

  Résultat : p

Un tri par sélection

- Code principal :

```
def trier(t): # Destructif : t est modifié
    for j in range(len(t)-1, 0, -1):
        i0 = pos_maxi_up_to(t, j)
        t[j], t[i0] = t[i0], t[j]
```

- Code secondaire (trouver l'indice...)

```
def pos_maxi_up_to(t, j): # jusqu'à j inclus
    pmp = 0
    for i in range(1, 1+j):
        if t[i] > t[pmp]:
            pmp = i
    return pmp
```

- Complexité ? Triangulaire !

L'algorithme d'Euclide

- $pgcd(a,b) = p_1^{\gamma_1} \ldots p_k^{\gamma_k} \ldots$
- Exemple :

$$pgcd(1386,840) = pgcd(2 \times 3^2 \times 7 \times 11, 2^3 \times 3 \times 5 \times 7)$$
$$= 2 \times 3 \times 7 = 42$$

- Nécessite d'avoir la factorisation en nombres premiers (hard)
- Algorithme d'Euclide :

$$a \geq b > 0 \qquad \Longrightarrow \qquad pgcd(a,b) = pgcd(b, a \,\%\, b)$$

- Exemple :

$$pgcd(1386,840) = pgcd(840,546) = pgcd(546,294)$$

$$= pgcd(294,252) = pgcd(252,42) = pgcd(42,0) = 42$$

L'algorithme d'Euclide

- Algorithme :

  Entrées : a_0, b_0

  $(a, b) \leftarrow (\max(a_0, b_0), \min(a_0, b_0))$ **tant que** $b > 0$ **faire**

  $\quad (a, b) \leftarrow (b, a\%b)$

  Résultat : a

- Code :

```
def euclide(a0, b0):
    a, b = max(a0,b0), min(a0,b0)
    while b>0:
        a, b = b, a%b
    return a
```

- Complexité : si $a, b \leq 2^n$, alors il y aura :
  - au plus n divisions euclidiennes, donc $O(n^3)$ opérations élémentaires.
  - au plus $2n$ opérations $l \leftarrow l - \alpha r$ dans les divisions euclidiennes, donc $O(n^2)$ opérations élémentaires !

C’est fini

Merci de votre attention !

Désolé, mais...

Impossible de passer à côté !