Algorithmes récursifs
Diviser pour régner

Michel Van Caneghem

Janvier 2003

Multiplication de matrices

Problème classique : $C = A \times B$ **avec** A **et** B **matrices carrés d'ordre** n**.**

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

```
for i:=1 to N do
    for j:=1 to N do begin
        C[i,j] := 0;
        for k:=1 to N do
            C[i,j] := C[i,j] + A[i,k]*B[k,j];
    end;
```

Complexité : $O(N^3)$

Multiplication de matrices (2)

On pensait ne pas pouvoir faire mieux. En 1968 une autre méthode est proposée par Strassen. On regarde d'abord la multiplication de matrices d'ordre 2 :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix}$$

$$= \begin{pmatrix} x_1+x_2-x_4+x_6 & x_4+x_5 \\ x_6+x_7 & x_2-x_3+x_5-x_7 \end{pmatrix}$$

$$\begin{array}{ll} x_1 = (b-d)(g+h) & x_5 = a(f-h) \\ x_2 = (a+d)(e+h) & x_6 = d(g-e) \\ x_3 = (a-c)(e+f) & x_7 = (c+d)e \\ x_4 = (a+b)h & \end{array}$$

Multiplication de matrices (3)

Il faut donc 7 multiplications et 18 additions au lieu de 8 multiplications et 4 additions.

C'est intéressant si le temps d'une multiplication est beaucoup plus grand que celui d'une addition.

Mais si nous cherchons à appliquer cette remarque sur les matrices alors il est évident que l'addition de 2 matrices est beaucoup plus simple et rapide ($O(N^2)$) que la multiplication de 2 matrices ($O(N^3)$).

Multiplication de matrices (4)

Supposons que la taille des matrice soit une puissance de 2 : $N = 2^n$, alors on peut toujours diviser récursivement les matrices en 4 de la manière suivante :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ avec } A_{ij} \text{ de taille } \frac{N}{2} = 2^{n-1}$$

On peut alors écrire un programme de multiplication de 2 matrices qui utilisera les règles précédentes. Il utilisera les deux procédures suivantes :

```
C = ProdMat(A, B, N), C = AddMat(A, B, N)
```

Multiplication de matrices (5)

```
fonction ProdMat(A, B, N)
begin
  if N = 1 then ProdMat:= A * B
  else begin Découper_en_4(A); Découper_en_4(B); end;
  X1 := ProdMat(SubMat(A12,A22,N/2),
                AddMat(B21,B22,N/2),N/2);
  ...
  X7 :=
  ...
  C22 := AddMat(SubMat(X2,X3,N/2),SubMat(X5,X7,N/2),N/2);
  Recompose(C);
end;
```

`ProdMat(...,N)` **fait donc 7 appels à** `ProdMat(...,N/2)` **et 18 appels à** `AddMat(...,N/2)`**.**

Multiplication de matrices (6)

Calcul du nombre de multiplications : **Avec** $N = 2^n$ **on a** $\ln N = n \times \ln 2$**. On trouve alors :**

$$n_{mult} = 7^n = 7^{\frac{\ln N}{\ln 2}} = e^{\ln 7 \frac{\ln N}{\ln 2}} = N^{\frac{\ln 7}{\ln 2}} = N^{2.807}$$

Calcul du nombre d'additions : **C'est un peu plus compliqué :**

$$\begin{aligned} a_n &= 7a_{n-1} + 18 \times (N/2)^2 \\ &= 7a_{n-1} + \frac{9}{2} \times 4^n \end{aligned}$$

On pose alors $a_n = 7^n y_n$**.**

Multiplication de matrices (7)

$$
\begin{aligned}
y_n &= y_{n-1} + \frac{9}{2} \times \left(\frac{4}{7}\right)^n \\
&= \frac{9}{2} \sum_{k=1}^{n} \left(\frac{4}{7}\right)^k \\
&\leq \frac{9}{2} \sum_{k=1}^{\infty} \left(\frac{4}{7}\right)^k = \frac{9}{2} \times \frac{1}{1 - 4/7} = \frac{21}{2} \\
y_n &\leq \frac{21}{2}
\end{aligned}
$$

et on trouve donc $n_{add} = O(7^n) = O(N^{2.807})$

Multiplication de matrices (8)

En conclusion, le produit de matrice (et la résolution d'un système linéaire) peut être fait en $O(N^{2.807})$ opérations.

- ✔ **On connaît des méthodes ou l'exposant est 2,5 et on pense que la valeur optimale est $2 + \epsilon$.**
- ✔ **De toutes façon la complexité minimale est en $O(N^2)$ car il faut pouvoir traiter $2N^2$ coefficients.**
- ✔ **Ce résultat présente surtout un intérêt théorique, car on a négligé le surcoût des opérations secondaires (Appels récursifs, découpages de matrices). Cette méthode n'est probablement intéressante que pour de très grosses matrices**

La résolution des systèmes linéaires

Les résultats précédents supposaient que le temps d'une multiplication était constant. Ce n'est pas le cas : cela dépend de la taille des nombres manipulés. Nous avons alors les résultats suivants :

- **Gauss en flottant :** $O(n^3)$.
- **Gauss en précision parfaite :** $O(n^5 \log^2 n)$.
- **Méthode modulaire :** $O(n^4 \log n)$.
- **Méthode P-Adique** $O(n^3 \log^2 n)$.

Multiplication de polynômes

$$P = \sum_{i=0}^{m} a_i x^i \qquad Q = \sum_{i=0}^{n} b_i x^i$$

$$PQ = \sum_{i=0}^{m+n} c_i x^i \quad \textbf{avec} \quad c_k = \sum_{i+j=k} a_i b_j$$

d'ou le programme

```
for (i = 0; i <= m + n; i++) C[i] = 0;
for (i = 0; i <= m; i++)
    for (j = 0; j <= n; j++) C[i+j] += A[i]*B[j];
```

et sa complexité en $O(mn)$.

Multiplication de polynômes (2)

Même remarque que pour les matrices : On multiplie 2 polynômes de degré $N = 2^n$.

$$P = P_1 + x^{N/2} P_2 \qquad Q = Q_1 + x^{N/2} Q_2$$

$$R_1 = P_1 \times Q_1 \quad R_2 = P_2 \times Q_2 \quad R_3 = (P_1 + P_2)(Q_1 + Q_2)$$

$$P \times Q = R_1 + (R_3 - R_2 - R_1) x^{N/2} + R_2 x^N$$

Il faut donc 3 multiplications au lieu de 4 (et 6 additions). On en déduit que :

$$C(N) = 3^n = N^{\frac{\log 3}{\log 2}} = N^{1.58}$$

Intéressant pour la multiplication de grands nombres.

Multiplication de polynômes (3)

Ce résultat est intéressant, mais on sait faire mieux !

Remarquons qu'un polynôme de degré n est entièrement défini par sa valeur en $n+1$ points.

Donc pour calculer $R = P \times Q$ il suffit de calculer :

$$P(x_i) \quad i \in [0..2n] \qquad Q(x_i) \quad i \in [0..2n]$$

$$R(x_i) = P(x_i) \times Q(x_i) \quad i \in [0..2n]$$

Mais il faut encore $O(n^2)$ multiplications !

Schéma de Horner

On peut évaluer la valeur d'un polynôme de degré N en un point avec N multiplications et N additions.

```
double value(C,n,x) {
    value = C[n];
    for (i=n-1; i>=0; i--) value=value*x+C[i]:
    return value;
}
```

Ex : $x^3 + 2x^2 + 3x + 4$ **au point 2**

BIEN : $2 \times (2 \times (2 \times 1 + 2) + 3) + 4$

MAL : $2 \times 2 \times 2 + 2 \times 2 \times 2 + 3 \times 2 + 4$

Multiplication de polynômes (4)

Il est facile de passer des coeffcients aux valeurs en $n+1$ points. Pour passer des valeurs aux coefficients, il faut résoudre un système linéaire de dimension $n+1$.

Il faut donc choisir un bon ensemble de points. ***Une brillante idée consiste à utiliser les racines nièmes de l'unité :***

Ex : Racine quatrièmes :

$$z^4 = 1 \qquad \{1, i, -1, -i\}$$

De manière générale :

$$\omega_k = e^{\frac{2\pi i k}{n}} \qquad k \in [0..n-1]$$

Transformée de Fourrier

Pour calculer la transformée de Fourrier de la séquence :

$$a_0, a_1, \ldots, a_{n-1}$$

On forme le polynôme de coefficients a_i et on calcule ses valeurs pour les racines nièmes de l'unité. Cette suite de valeurs est *La Transformée de Fourrier* de la séquence :

$$f(t) = a_0 + a_1 t + \cdots + a_{n-1} t^{n-1}$$

$$f(\omega_j) = \sum_{k=0}^{n-1} a_k \omega_j^k = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}$$

L'idée géniale

Toujours la même : couper le problème en deux. Cette fois on va utiliser les termes de rangs pairs et ceux de rangs impairs. Si n pair :

$$f(\omega_j) = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}} \quad k = 2m \text{ **ou** } k = 2m+1$$

$$f(\omega_j) = \sum_{m=0}^{n/2-1} a_{2m} e^{\frac{2\pi ij2m}{n}} + \sum_{m=0}^{n/2-1} a_{2m+1} e^{\frac{2\pi ij(2m+1)}{n}}$$

$$f(\omega_j) = \sum_{m=0}^{n/2-1} a_{2m} e^{\frac{2\pi ijm}{n/2}} + e^{\frac{2\pi ij}{n}} \sum_{m=0}^{n/2-1} a_{2m+1} e^{\frac{2\pi ijm}{n/2}}$$

L'idée géniale (2)

Pour résumer pour calculer la transformée de Fourrier de taille N on calcule :

- ✗ **La transformée de Fourrier ($N/2$) des coeffiocients pairs**
- ✗ **La transformée de Fourrier ($N/2$) des coefficients impairs**

Il reste un petit problème pour passer de $N/2$ valeurs à N valeurs. On profite de la périodicité de la transformée de Fourrier. Si $J = j + N/2$ alors :

$$f(\omega_J) = \sum_{m=0}^{N/2-1} a_m e^{\frac{2\pi i(j+N/2)m}{N/2}} = e^{2\pi im} \sum_{m=0}^{N/2-1} a_m e^{\frac{2\pi ijm}{N/2}}$$

Transformée de Fourrier (2)

```
FFT(A,FA,N) {
    if (N == 1) {FA[0] = A[0]; return;}
    for (i = 0; i < N/2; i ++)
        {Pair[i]=A[2*i]; Impair[i]=A[2*i+1];}
    FFT(Pair,FPair,N/2);
    FFT(Impair,FImpair,N/2);
    for (j = 0; j < N; j++) {
        omega = exp(2*Pi*I*j/N);
        FA[j]=FPair[j%N/2]+omega*FImpair[j%N/2];
    }
}
```

Transformée de Fourrier (3)

Examinons maintenant la complexité :

$$T(N) = 2T(N/2) + N$$

$$T(N) = O(N \log N)$$

Il s'agit bien sur d'opération sur les nombres complexes.

Cet algorithme est appelé Transformée de Fourrier rapide ou FFT (Fast Fourrier Transform). Cet algorithme est d'une importance capitale pour le traitement du signal. Il a été inventé en 1965 par Cooley et Tukey

Multiplication de polynômes (5)

On a vu que la première phase : le calcul des valeurs, peut se faire avec une complexité de $O(N \log N)$

Pour Passer des valeurs aux coefficients on peut remarquer que :

$$f(\omega_j) = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}} \qquad a_j = \frac{1}{N} \sum_{k=0}^{n-1} f(\omega_k) e^{\frac{-2\pi ijk}{n}}$$

① **Prendre le conjugé de la suite des valeurs ;**

② **Appliquer la procédure FFT ;**

③ **Prendre le conjugé du résultat et le diviser par N**

La complexité est encore de $O(N \log N)$

Multiplication de polynômes (6)

Pour multiplier deux polynômes P et Q de degré $N-1$:

❶ **Calculer les valeurs de P pour les racines $2N-1$ ième de l'unité ;**

❷ **Calculer les valeurs de Q pour les racines $2N-1$ ième de l'unité ;**

❸ **Multiplier ces valeurs ;**

❹ **Appliquer la FFT pour calculer les coefficients de PQ.**

La complexité est bien de $O(N \log N)$

C'est la méthode de Schönhage et Strassen (1971) pour multiplier des grands nombres en un temps $O(N \log N)$.

Multiplication des grands nombres

- ✔ **Multiplication normale :** $O(n^2)$ **;**
- ✔ **En divisant par 2 (Karatsuba) :** $O(n^{1.585})$ **;**
- ✔ **Algorithme de Toom - Cook :** $O(n2^{3.5\sqrt{\log n}})$ **;**
- ✔ **Méthode de Schönhage et Strassen :** $O(n \log n \log \log n)$ **;**
- ✔ **Arithmétique modulaire (borné) :** $O(n)$**.**

On peut remarquer que :

$$n2^{3.5\sqrt{\log n}} = n^{1+3.5/\sqrt{\log n}}$$

qui bien sûr, croit plus vite que $\log n$ **! ! !**

Conclusion

- **L'analyse des algorithmes est souvent complexe ;**
- ***Mais les algorithmes obtenus sont très performants*** **;**
- **Utilisez des algorithmes simples et réguliers quand les performances ne sont pas nécessaires ;**
- **Méthode diviser pour régner :**

1. **Diviser le problème en sous-problèmes ;**
2. **Trouver une méthode pour résoudre le problème à partir des sous problèmes.**

Algorithmes sur les chaînes

Michel Van Caneghem

Janvier 2003

Recherche de motifs

Opération très courante en Informatique. Sous Unix on trouve les outils suivants :

- ✔ **grep : recherche de motifs avec expression régulière partielles,**
- ✔ **egrep : idem avec expression régulières étendues,**
- ✔ **fgrep : fast grep (chaînes fixes),**
- ✔ **bm : search a file for a string,**
- ✔ **gre : grep amélioré pour chaînes multiples,**
- ✔ **e ?grep : version de GNU de egrep (plus rapide),**
- ✔ **agrep : approximate grep,**
- ✔ **...**

Algorithme naïf

Vocabulaire : On cherche un motif (sous-chaîne) x de longueur m dans un texte t de longueur n. En général le texte est beaucoup plus grand que le motif. Ces textes sont écrits dans un vocabulaire V de q caractères.

```
i = 1; j = 1;
while (i <= m && j <= n) do
    if (t[j] == x[i]) i++; j++;
    else j = j - i + 2; i = 1;
if (i > m) then "motif trouve en j - m"
else "pas d'occurrences"
```

Algorithme naïf (2)

Complexité : Dans le pire des cas $O(mn)$. Il suffit de choisir : $x = a^{m-1}b$ et $t = a^{n-1}b$ et on trouve un nombre de comparaison de : $(n - m + 1) * m$.
Si l'alphabet a q caractère, la complexité en moyenne est :

$$(1 + \frac{1}{q-1})n \leq 2n$$

si les caractères sont uniformément distribués dans le texte.
En pratique pour rechercher une chaîne de 7 chiffres parmi 100000 chiffres (les décimales de $\sqrt{2}$) j'ai fait 111267 comparaisons (th : 111111).

Algorithme de Knuth Morris Pratt

Aspect historique : En 1970, Cook avait prouvé de manière théorique que l'on pouvait faire la recherche d'un motif dans le pire des cas en $O(m+n)$, sans donner d'algorithme. En reprenant ce travail, Knuth et Pratt réussirent à exhiber un algorithme ayant cette complexité. Morris avait découvert le même algorithme de manière indépendante.

L'idée consiste à remarquer que si l'on a un échec à la $i$ème lettre du motif alors :

$$x_1 \ldots x_{i-1} = t_{k+1} \ldots t_{k+i-1} \quad \textbf{et} \quad x_i \neq t_{k+i}$$

On connaît donc les $i-2$ prochains caractères du texte quand on examinera le texte à partir de la position $k+2$.

Algorithme de Knuth Morris Pratt (2)

Voyons un exemple (Morris Pratt) :

```
motif : ABACABAC
texte : B   A   B   A   C   A   C   A   B   A   C   A   A   B
        A   .
            A   B   A   C   A   B   .
                            A   B   .
                                A   .
                                    A   B   A   C   A   B   .
                                                        A   B
```

Il faut 16 comparaisons. On peut alors construire la table suivante :

```
        A   B   A   C   A   B   A   C
s(i) =  0   1   1   2   1   2   3   4
```

Algorithme de Knuth Morris Pratt (3)

```
i = 1; j = 1;
while (i <= m && j <= n) do
    if (i >= 1 && t[j] != x[i]) i = s[i]
    else {i++; j++;}
if (i > m) then "motif trouve en j - m"
else "pas d'occurrences"
```

Remarque1 : Maintenant `j` **ne recule plus.**

Remarque2 : On peut enlever le test `j <= n` **en recopiant le motif à la fin du texte, comme cela on trouve toujours le motif.**

Algorithme de Knuth Morris Pratt (4)

On peut montrer que le nombre de comparaisons dans le pire des cas est borné par $2n$.

La complexité en moyenne est donnée par

$$(1 + \frac{1}{q-1} - \frac{2}{q^2})n$$

Avec une chaîne de 7 chiffres et un texte de 100000 chiffres on trouve : 109111 comparaisons au lieu de 111111.

Ce résultat est très décevant et personne n'utilise cet algorithme qui n'avait qu'un intérêt théorique.

Algorithme de Boyer et Moore

L'idée consiste également à prétraiter le motif. Mais cette fois on fait l'analyse du motif de la droite vers la gauche. Voyons un exemple :

```
motif : OBELIX
texte : J E C H E R C H E U N G A U L O I S
        . . . . . X
              . . . . . X
                    . . . . . X
Nombre de comparaisons : 3
```

Algorithme de Boyer et Moore (2)

Algorithme de Horspool **Quand on compare le motif au texte, en fonction de la lettre du texte ayant induit la différence, on décale le motif vers la droite de manière à faire coïncider cette lettre avec l'occurrence la plus à droite de la même lettre dans le motif.**

Il faut donc définir une *fonction de dernière occurrence* qui indique de combien on doit décaler le motif. Par exemple :

```
        O  B  E  L  I  X  *
d =     5  4  3  2  1  6  6
```

Algorithme de Boyer et Moore (3)

```
j = m;
while (j <= n) do {
    i = m;
    while (i > 0 && t[j-m+i] == x[i]) do i--;
    if i == 0 then "motif trouve en j - m"
    else j = j + d[t[j]];
    }
```

Algorithme de Boyer et Moore (4)

On peut améliorer cet algorithme en étudiant mieux les suffixes. La complexité dans le pire des cas est de $O(mn)$, mais en moyenne on trouve une complexité de

$$\frac{2}{q+1}n$$

ou souvent n/m comparaisons. En reprenant le même exemple on trouve 18181 comparaisons (soit 5 fois moins).

C'est cet algorithme amélioré qui est à la base de tous les outils de recherche rapide du style egrep, fgrep, agrep, ... (qui sont d'une rapidité stupéfiante : 3,3s (45s en tout) pour chercher une chaîne dans un texte de 630Mo : `grep archeopterix grosFichier`**).**

Algorithme de Rabin-Karp

L'idée de base (1980) consiste a utiliser une fonction de "hash-code" pour vérifier l'égalité des chaînes.

On calcule le hash-code de la chaîne à rechercher et celui (h_i) de la sous-chaîne de longueur m commençant en position i. Si il y a égalité on compare les chaînes, sinon on incrémente i de 1 et on recommence.

Cela parait plus coûteux de calculer un hash-code que comparer des caractères, *sauf si il y a un moyen simple de passer de h_{i-1} à h_i*.

Algorithme de Rabin-Karp (2)

A chaque caractère on fait correspondre un code compris entre 0 et $d-1$ et on calcule :

$$h_i = \sum_{k=0}^{m-1} code(t[i+k])d^{m-1-k} \pmod q$$

q **étant un nombre premier le plus grand possible. On remarque alors que :**

$$h_{i+1} = d \times h_i + code(t[i+m]) - d^m code(t[i]) \pmod q$$

On peut calculer une fois pour toutes $d^m \pmod q$. Pour éviter les problèmes de débordement, il faut pouvoir calculer $(d+1)q$. Un bon choix serait de prendre $d = 32$ (pour faire les multiplications par décalage) et par exemple $q = 33554393$.

Algorithme de Rabin-Karp (3)

La complexité dans le pire des cas est $O(mn)$, mais si q est bien choisi on peut espérer une complexité linéaire : $O(m+n)$.

L'intérêt de cet algorithme est de montrer que l'on a pu trouver un algorithme très simple à programmer ayant une complexité linéaire.

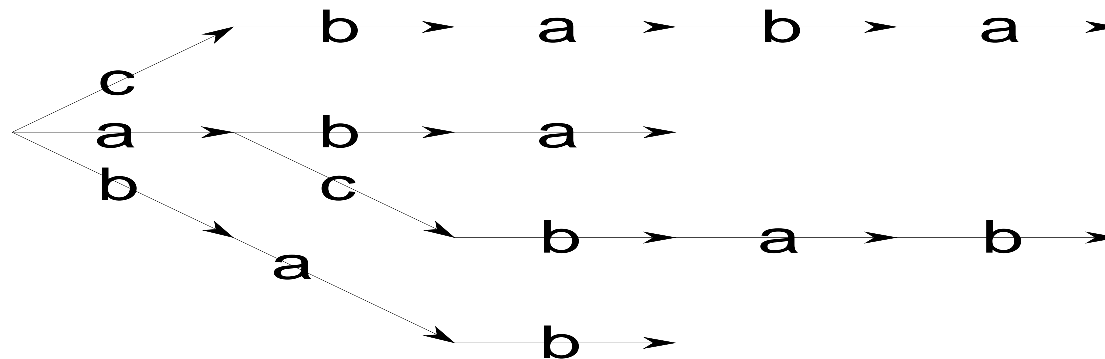
D'un point de vue pratique, sauf si l'on fait des recherches multiples sur des chaînes ayant les mêmes longueurs, cet algorithme semble n'avoir aucun intérêt spécial.

Recherches multiples

On veut rechercher en même temps plusieurs (p) motifs dans un texte. La première idée consiste à appliquer p fois un des algorithmes précédents. Cela implique qu'il faut passer p fois sur le texte.

Cela peut être amélioré en codant l'ensemble des motifs en partie commune et en appliquant par exemple l'algorithme naïf.

$$X = \{aba, bab, acb, acbab, cbaba\}$$



Recherches multiples (2)

Si on utilise comme algorithme de base Knuth-Morris-Pratt on obtient alors l'algorithme de Aho et Corasick (1975) qui est utilisé par `fgrep`**.**

Si on utilise Boyer-Moore on obtient l'algorithme de Commentz-Walter (1979) qui est plus rapide.

Mais toutes les idées sont bonnes : **On peut par exemples découper tous les motifs à rechercher en tranches de b caractères. Pour chaque tranche on calcule son hash-code. Ensuite on lit le texte par groupe de b caractères. Cela consiste à augmenter la taille du vocabulaire. On peut ensuite utiliser une des méthodes précédentes (utilisé dans** `agrep`**).**

Recherche par expression régulière

On veut chercher dans un texte un ensemble de motifs ayant certaines propriétés. Un outil très puissant consiste à utiliser les expressions régulières qui sont construites à partir des opérations suivantes :

- **la concaténation : Ex : AB l'expression A suivie de l'expression B,**
- **l'union : Ex : $A|B$ ou bien l'expression A ou bien l'expression B,**
- **la fermeture : Ex : A^* l'expression A répété un nombre arbitraire de fois (y compris 0).**

Par exemple : `((a | b | c)[0-9]* | _ )`

Recherche par expression régulière (2)

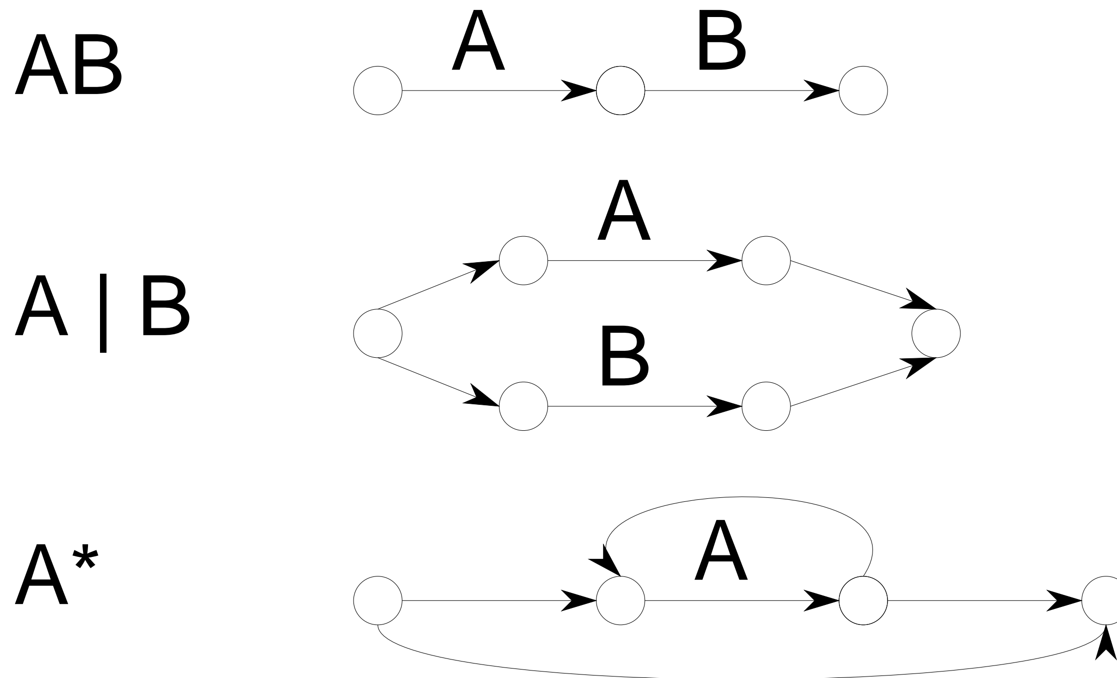
On sait que le langage reconnu par une expression régulière est le même que celui reconnu par un automate d'état finis. Deux stratégies sont possibles :

- **Construire l'automate déterministe reconnaissant l'expression régulière. Cela sera très efficace, mais on risque d'avoir un très grand nombre d'état.**
- **Construire un automate non-déterministe (donc moins efficace) mais dont le nombre d'état est proportionnel à la taille de l'expression régulière.**

C'est en général le deuxième choix que l'on fait.

Recherche par expression régulière (3)

Théorème : Pour toute expression rationnelle e de taille m, il existe un automate normalisé reconnaissant $L(e)$, et dont le nombre d'états est au plus $2m$.



Recherche par expression régulière (4)

La complexité de la recherche est en $O(mn)$, si m est la taille de l'expression régulière. (`grep`).

`grep` **permet aussi de remplacer un motif par un autre. Par exemple si on veut permuter les argument de $log(x, b)$ alors on peut écrire :**

```
test = log(7,2);
test = log(x+y,b*3);
> sed 's/log(\([^,]*\),\([^)]*\))/log(\2,\1)/g'
test = log(2,7);
test = log(b*3,x+y);
```