

Algorithme de Knuth-Morris-Pratt

Serge Iovleff

le problème...

- On veut trouver les occurrences d'une chaîne P dans un texte S .
- Par exemple :
 - Trouver la chaîne $P = aaaaaab$ dans le texte
 $T = aab$
 - Trouver la chaîne $P = abaabbab$ dans le texte
 $T = ababbabaabaabbababbabaabbab$
- En minimisant le nombre de comparaison.

le problème...

- On veut trouver les occurrences d'une chaîne P dans un texte S .
- Par exemple :
 - Trouver la chaîne $P = aaaaab$ dans le texte
 $T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$
 - Trouver la chaîne $P = abaabbab$ dans le texte
 $T = ababbabaabaabbababbabaabbab$
- En minimisant le nombre de comparaison.

le problème...

- On veut trouver les occurrences d'une chaîne P dans un texte S .
- Par exemple :
 - Trouver la chaîne $P = aaaaab$ dans le texte
 $T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$
 - Trouver la chaîne $P = abaabbab$ dans le texte
 $T = ababbabaabaabbababbabaabbab$
- En minimisant le nombre de comparaison.

le problème...

- On veut trouver les occurrences d'une chaîne P dans un texte S .
- Par exemple :
 - Trouver la chaîne $P = aaaaab$ dans le texte
 $T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$
 - Trouver la chaîne $P = abaabbab$ dans le texte
 $T = ababbabaabaabbababbabaabbab$
- En minimisant le nombre de comparaison.

Un Algorithme naïf

La chaîne P peut être trouvée dans le texte S avec l'algorithme suivant :

- ① Fixer $i = 1$
- ② Tant qu'il reste des positions à vérifier
 - Comparer lettre à lettre la chaîne P et le texte S à partir de la position i
 - Si la chaîne correspond, terminer le traitement et retourner i comme position du début de l'occurrence
 - Sinon fixer $i = i + 1$
- ③ Terminer le traitement, aucune occurrence n'a été trouvée.

Un Algorithme naïf et couteux

- Cet algorithme a un inconvénient, après une comparaison infructueuse, la comparaison suivante débutera à la position $i + 1$.
- Suposons que l'on recherche la chaîne $P = aaaaaaab$ dans le texte $T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$.
- On effectuera la suite de comparaisons

```

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
aaaaaaab
  aaaaaaab
    aaaaaaab
      aaaaaaab
        aaaaaaab
          ...

```

- Ce qui fera au total 8×26 comparaisons !

Un Algorithme naïf et couteux

- Cet algorithme a un inconvénient, après une comparaison infructueuse, la comparaison suivante débutera à la position $i + 1$.
- Suposons que l'on recherche la chaîne $P = aaaaaaab$ dans le texte $T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$.
- On effectuera la suite de comparaisons

```

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
aaaaaaab
  aaaaaaab
    aaaaaaab
      aaaaaaab
        aaaaaaab
          ...

```

- Ce qui fera au total 8×26 comparaisons !

Un Algorithme naïf et couteux

- Cet algorithme a un inconvénient, après une comparaison infructueuse, la comparaison suivante débutera à la position $i + 1$.
- Suposons que l'on recherche la chaîne $P = aaaaaaab$ dans le texte $T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$.
- On effectuera la suite de comparaisons

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab

aaaaaab

aaaaaab

aaaaaab

aaaaaab

aaaaaab

...

- Ce qui fera au total 8×26 comparaisons !

Un Algorithme naïf et couteux

- Cet algorithme a un inconvénient, après une comparaison infructueuse, la comparaison suivante débutera à la position $i + 1$.
- Suposons que l'on recherche la chaîne $P = aaaaaaab$ dans le texte $T = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab$.
- On effectuera la suite de comparaisons

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab

aaaaaaab

aaaaaaab

aaaaaaab

aaaaaaab

aaaaaaab

...

- Ce qui fera au total 8×26 comparaisons !

L'algorithme KPM

- L'*algorithme de Knuth-Morris-Pratt* (souvent abrégé par *algorithme KMP*) permettant de trouver les occurrences d'une chaîne P dans un texte S en effectuant un pré-traitement de la chaîne, qui fournit une information suffisante pour déterminer où continuer la recherche en cas de non-correspondance.
- Cela permet à l'algorithme de ne pas ré-examiner les caractères qui ont été précédemment vérifiés, et donc de limiter le nombre de comparaisons nécessaires.
- L'algorithme a été inventé par Donald Knuth, Vaughan Pratt, et indépendamment par J. H. Morris [Knuth] en 1975.

L'algorithme KPM

- *L'algorithme de Knuth-Morris-Pratt* (souvent abrégé par *algorithme KMP*) permettant de trouver les occurrences d'une chaîne P dans un texte S en effectuant un pré-traitement de la chaîne, qui fournit une information suffisante pour déterminer où continuer la recherche en cas de non-correspondance.
- Cela permet à l'algorithme de ne pas ré-examiner les caractères qui ont été précédemment vérifiés, et donc de limiter le nombre de comparaisons nécessaires.
- L'algorithme a été inventé par Donald Knuth, Vaughan Pratt, et indépendamment par J. H. Morris [Knuth] en 1975.

L'algorithme KPM

- *L'algorithme de Knuth-Morris-Pratt* (souvent abrégé par *algorithme KMP*) permettant de trouver les occurrences d'une chaîne P dans un texte S en effectuant un pré-traitement de la chaîne, qui fournit une information suffisante pour déterminer où continuer la recherche en cas de non-correspondance.
- Cela permet à l'algorithme de ne pas ré-examiner les caractères qui ont été précédemment vérifiés, et donc de limiter le nombre de comparaisons nécessaires.
- L'algorithme a été inventé par Donald Knuth, Vaughan Pratt, et indépendamment par J. H. Morris [Knuth] en 1975.

Un exemple : Notations

- Pour présenter le principe de fonctionnement de l'algorithme, un exemple particulier est considéré : la chaîne P vaut ABCDABD et le texte S est ABC ABCDAB ABCDABCDABDE.
- Pour représenter les chaînes de caractères, on utilisera des tableaux dont les indices débutent à zéro. Ainsi, le C de la chaîne P sera notée $P[2]$.
- m désigne la position dans le texte S à laquelle la chaîne P est en cours de vérification, et i la position du caractère actuellement vérifié dans P .

Un exemple : Notations

- Pour présenter le principe de fonctionnement de l'algorithme, un exemple particulier est considéré : la chaîne P vaut ABCDABD et le texte S est ABC ABCDAB ABCDABCDABDE.
- Pour représenter les chaînes de caractères, on utilisera des tableaux dont les indices débutent à zéro. Ainsi, le C de la chaîne P sera notée $P[2]$.
- m désigne la position dans le texte S à laquelle la chaîne P est en cours de vérification, et i la position du caractère actuellement vérifié dans P .

Un exemple : Notations

- Pour présenter le principe de fonctionnement de l'algorithme, un exemple particulier est considéré : la chaîne P vaut ABCDABD et le texte S est ABC ABCDAB ABCDABCDABDE.
- Pour représenter les chaînes de caractères, on utilisera des tableaux dont les indices débutent à zéro. Ainsi, le C de la chaîne P sera notée $P[2]$.
- m désigne la position dans le texte S à laquelle la chaîne P est en cours de vérification, et i la position du caractère actuellement vérifié dans P .

Un exemple (1)

- La situation de départ est la suivante

```

          1          2
01234567890123456789012
m : v
S : ABC ABCDAB ABCDABCDABDE
P : ABCDABD
i : ^
0123456

```

- L'algorithme teste la correspondance des caractères les uns après les autres. Ainsi, à la quatrième étape, $m = 0$ et $i = 3$. $S[3]$ est un espace et $P[3] = 'D'$, la correspondance n'est pas possible.

```

          1          2
01234567890123456789012
m : v
S : ABC ABCDAB ABCDABCDABDE
P : ABCDABD
i :   ^
0123456

```

Un exemple (1)

- La situation de départ est la suivante

```

                1         2
01234567890123456789012
m : v
S : ABC ABCDAB ABCDABCDABDE
P : ABCDABD
i : ^
0123456

```

- L'algorithme teste la correspondance des caractères les uns après les autres. Ainsi, à la quatrième étape, $m = 0$ et $i = 3$. $S[3]$ est un espace et $P[3] = 'D'$, la correspondance n'est pas possible.

```

                1         2
01234567890123456789012
m : v
S : ABC ABCDAB ABCDABCDABDE
P : ABCDABD
i :      ^
0123456

```

Un exemple (2)

- Plutôt que de recommencer avec $m = 1$, l'algorithme sait qu'aucun A n'est présent dans S entre les positions 1 et 3. De ce fait, l'algorithme avance jusqu'au caractère suivant, en posant $m = 4$ et $i = 0$.

```

          1         2
01234567890123456789012
m :      v
S : ABC ABCDAB ABCDABCDABDE
P :      ABCDABD
i :      ^
          0123456

```

- Une correspondance presque complète est rapidement obtenue quand, avec $i = 6$, la vérification échoue.

```

          1         2
01234567890123456789012
m :      v
S : ABC ABCDAB ABCDABCDABDE
P :      ABCDABD
i :      ^
          0123456

```

Un exemple (2)

- Plutôt que de recommencer avec $m = 1$, l'algorithme sait qu'aucun A n'est présent dans S entre les positions 1 et 3. De ce fait, l'algorithme avance jusqu'au caractère suivant, en posant $m = 4$ et $i = 0$.

```

                1         2
01234567890123456789012
m :           v
S : ABC ABCDAB ABCDABCDABDE
P :         ABCDABD
i :           ^
           0123456

```

- Une correspondance presque complète est rapidement obtenue quand, avec $i = 6$, la vérification échoue.

```

                1         2
01234567890123456789012
m :           v
S : ABC ABCDAB ABCDABCDABDE
P :         ABCDABD
i :           ^
           0123456

```

Un exemple (3)

- Juste avant la fin de cette correspondance partielle, l'algorithme est passé sur le motif AB, qui pourrait correspondre au début d'une autre correspondance. Ainsi, l'algorithme reprend son traitement au caractère courant, avec $m = 8$ et $i = 2$.

```

                1           2
01234567890123456789012
m :           v
S : ABC ABCDAB ABCDABCDABDE
P :           ABCDABD
i :           ^
                0123456

```

- Cette vérification échoue immédiatement. L'algorithme poursuit la recherche avec $m = 11$ et en réinitialisant $i = 0$.

```

                1           2
01234567890123456789012
m :           v
S : ABC ABCDAB ABCDABCDABDE
P :           ABCDABD
i :           ^
                0123456

```

Un exemple (3)

- Juste avant la fin de cette correspondance partielle, l'algorithme est passé sur le motif AB, qui pourrait correspondre au début d'une autre correspondance. Ainsi, l'algorithme reprend son traitement au caractère courant, avec $m = 8$ et $i = 2$.

```

                1           2
01234567890123456789012
m :             v
S : ABC ABCDAB ABCDABCDABDE
P :             ABCDABD
i :             ^
                0123456

```

- Cette vérification échoue immédiatement. L'algorithme poursuit la recherche avec $m = 11$ et en réinitialisant $i = 0$.

```

                1           2
01234567890123456789012
m :             v
S : ABC ABCDAB ABCDABCDABDE
P :             ABCDABD
i :             ^
                0123456

```

Un exemple (4)

- À nouveau, l'algorithme trouve une correspondance partielle ABCDAB, mais le caractère suivant C ne correspond pas au caractère final D de la chaîne.

```

                1      2
01234567890123456789012
m :                v
S : ABC ABCDAB ABCDABCDABDE
P :                ABCDABD
i :                ^
                0123456

```

- ```

 1 2
01234567890123456789012
m : v
S : ABC ABCDAB ABCDABCDABDE
P : ABCDABD
i : ^
 0123456

```

# Un exemple (4)

- À nouveau, l'algorithme trouve une correspondance partielle ABCDAB, mais le caractère suivant C ne correspond pas au caractère final D de la chaîne.

```

 1 2
01234567890123456789012
m : v
S : ABC ABCDAB ABCDABCDABDE
P : ABCDABD
i : ^
 0123456

```

- ```

                1      2
01234567890123456789012
m :                v
S : ABC ABCDAB ABCDABCDABDE
P :                ABCDABD
i :                ^
                0123456

```


Un exemple (5)

- Avec le même raisonnement que précédemment, l'algorithme reprend avec $m = 15$, pour démarrer à la chaîne de deux caractères AB menant à la position courante, en fixant $i = 2$, nouvelle position courante.

```

                1         2
01234567890123456789012
m :                ' ' v ' ' '
S : ABC ABCDAB ABCDABCDABDE
P :                ABCDABD
i :                ' ' ' ~ ' ' '
                0123456

```

- Cette fois, la correspondance est complète, l'algorithme retourne la position $m = 15$ comme origine.

```

                1         2
01234567890123456789012
m :                v
S : ABC ABCDAB ABCDABCDABDE
P :                ABCDABD
i :                ^
                0123456

```

Un exemple (5)

- Avec le même raisonnement que précédemment, l'algorithme reprend avec $m = 15$, pour démarrer à la chaîne de deux caractères **AB** menant à la position courante, en fixant $i = 2$, nouvelle position courante.

```

                1         2
01234567890123456789012
m :                ' ' v ' ' '
S : ABC ABCDAB ABCDABCDABDE
P :                ABCDABD
i :                ' ' ' ~ ' ' '
                0123456

```

- Cette fois, la correspondance est complète, l'algorithme retourne la position $m = 15$ comme origine.

```

                1         2
01234567890123456789012
m :                v
S : ABC ABCDAB ABCDABCDABDE
P :                ABCDABD
i :                ^
                0123456

```

Principe de l'algorithme

- L'algorithme suppose l'existence d'un tableau donnant les “correspondances partielles” (décrit plus loin), indiquant où chercher le début potentiel de la prochaine occurrence, dans le cas où la vérification de l'occurrence potentielle actuelle échoue.
- ce tableau, désigné par T , peut être considéré comme une *boîte noire* ayant la propriété suivante : *si l'on dispose d'une correspondance partielle de $S[m]$ à $S[m + i - 1]$, mais qui échoue lors de la comparaison entre $S[m + i]$ et $P[i]$, alors la prochaine occurrence potentielle démarre à la position $m + i - T[i - 1]$.*
- En particulier, $T[-1]$ existe et est défini à -1 .

Principe de l'algorithme

- L'algorithme suppose l'existence d'un tableau donnant les “correspondances partielles” (décrit plus loin), indiquant où chercher le début potentiel de la prochaine occurrence, dans le cas où la vérification de l'occurrence potentielle actuelle échoue.
- ce tableau, désigné par T , peut être considéré comme une *boîte noire* ayant la propriété suivante : *si l'on dispose d'une correspondance partielle de $S[m]$ à $S[m + i - 1]$, mais qui échoue lors de la comparaison entre $S[m + i]$ et $P[i]$, alors la prochaine occurrence potentielle démarre à la position $m + i - T[i - 1]$.*
- En particulier, $T[-1]$ existe et est défini à -1 .

Principe de l'algorithme

- L'algorithme suppose l'existence d'un tableau donnant les “correspondances partielles” (décrit plus loin), indiquant où chercher le début potentiel de la prochaine occurrence, dans le cas où la vérification de l'occurrence potentielle actuelle échoue.
- ce tableau, désigné par T , peut être considéré comme une *boîte noire* ayant la propriété suivante : *si l'on dispose d'une correspondance partielle de $S[m]$ à $S[m + i - 1]$, mais qui échoue lors de la comparaison entre $S[m + i]$ et $P[i]$, alors la prochaine occurrence potentielle démarre à la position $m + i - T[i - 1]$.*
- En particulier, $T[-1]$ existe et est défini à -1 .

Algorithme KPM (en C)

```
int kmp_recherche(char *P, char *S)
{
    extern int T[];
    int m = 0;
    int i = 0;
    while (S[m + i] != '\0' && P[i] != '\0')
    {
        if (S[m + i] == P[i]){ ++i;}
        else
        { m += i - T[i];
          if (i > 0) i = T[i];
        }
    }
    if (P[i] == '\0') { return m;}
    else                { return m + i;}
}
```

Algorithme KPM (en C)

```
int kmp_recherche(char *P, char *S)
{
    extern int T[];
    int m = 0;
    int i = 0;
    while (S[m + i] != '\0' && P[i] != '\0')
    {
        if (S[m + i] == P[i]){ ++i;}
        else
        { m += i - T[i];
          if (i > 0) i = T[i];
        }
    }
    if (P[i] == '\0') { return m;}
    else                { return m + i;}
}
```

Algorithme KPM (en C)

```
int kmp_recherche(char *P, char *S)
{
    extern int T[];
    int m = 0;
    int i = 0;
    while (S[m + i] != '\0' && P[i] != '\0')
    {
        if (S[m + i] == P[i]){ ++i;}
        else
        { m += i - T[i];
          if (i > 0) i = T[i];
        }
    }
    if (P[i] == '\0') { return m;}
    else                { return m + i;}
}
```


Principe de construction

- L'observation-clé est qu'en ayant vérifié une partie du texte avec une "première portion" de la chaîne, il est possible de déterminer à quelles positions peuvent commencer les possibles occurrences qui suivent.
- Les *motifs* (les sous-parties de la chaîne) sont "pré-recherchés" dans la chaîne, et une liste est établie, indiquant toutes les positions possibles auxquelles continuer pour sauter un maximum de caractères inutiles
- Pour chaque position dans la chaîne, il faut donc déterminer la longueur du motif initial le plus long, qui se termine à la position courante.

Dans l'exemple précédent on obtient le tableau

i	-1	0	1	2	3	4	5	6
$P[i]$		A	B	C	D	A	B	D
$T[i]$	-1	0	0	0	0	1	2	0

Principe de construction

- L'observation-clé est qu'en ayant vérifié une partie du texte avec une "première portion" de la chaîne, il est possible de déterminer à quelles positions peuvent commencer les possibles occurrences qui suivent.
- Les *motifs* (les sous-parties de la chaîne) sont "pré-recherchés" dans la chaîne, et une liste est établie, indiquant toutes les positions possibles auxquelles continuer pour sauter un maximum de caractères inutiles
- Pour chaque position dans la chaîne, il faut donc déterminer la longueur du motif initial le plus long, qui se termine à la position courante.

Dans l'exemple précédent on obtient le tableau

i	-1	0	1	2	3	4	5	6
$P[i]$		A	B	C	D	A	B	D
$T[i]$	-1	0	0	0	0	1	2	0

Principe de construction

- L'observation-clé est qu'en ayant vérifié une partie du texte avec une "première portion" de la chaîne, il est possible de déterminer à quelles positions peuvent commencer les possibles occurrences qui suivent.
- Les *motifs* (les sous-parties de la chaîne) sont "pré-recherchés" dans la chaîne, et une liste est établie, indiquant toutes les positions possibles auxquelles continuer pour sauter un maximum de caractères inutiles
- Pour chaque position dans la chaîne, il faut donc déterminer la longueur du motif initial le plus long, qui se termine à la position courante.

Dans l'exemple précédent on obtient le tableau

i	-1	0	1	2	3	4	5	6
$P[i]$		A	B	C	D	A	B	D
$T[i]$	-1	0	0	0	0	1	2	0

Principe de construction

- L'observation-clé est qu'en ayant vérifié une partie du texte avec une "première portion" de la chaîne, il est possible de déterminer à quelles positions peuvent commencer les possibles occurrences qui suivent.
- Les *motifs* (les sous-parties de la chaîne) sont "pré-recherchés" dans la chaîne, et une liste est établie, indiquant toutes les positions possibles auxquelles continuer pour sauter un maximum de caractères inutiles
- Pour chaque position dans la chaîne, il faut donc déterminer la longueur du motif initial le plus long, qui se termine à la position courante.

Dans l'exemple précédent on obtient le tableau

i	-1	0	1	2	3	4	5	6
$P[i]$		A	B	C	D	A	B	D
$T[i]$	-1	0	0	0	0	1	2	0

Algorithme de construction des correspondances partielles (en C)

```
void kmp_tableau(char *P)
{
    extern int T[];
    int i = 0;
    int j = -1;
    char c = '\0';
    T[0] = j; /* = -1 */
    while (P[i] != '\0')
    {
        if (P[i] == c) { T[++i] = ++j; }
        else if (j > 0) { j = T[j]; }
        else { T[++i] = 0; j = 0; }
        c = P[j];
    }
}
```

Algorithme de construction des correspondances partielles (en C)

```
void kmp_tableau(char *P)
{
    extern int T[];
    int i = 0;
    int j = -1;
    char c = '\0';
    T[0] = j; /* = -1 */
    while (P[i] != '\0')
    {
        if (P[i] == c) { T[++i] = ++j; }
        else if (j > 0) { j = T[j]; }
        else { T[++i] = 0; j = 0; }
        c = P[j];
    }
}
```

}

Algorithme de construction des correspondances partielles (en C)

```
void kmp_tableau(char *P)
{
    extern int T[];
    int i = 0;
    int j = -1;
    char c = '\0';
    T[0] = j; /* = -1 */
    while (P[i] != '\0')
    {
        if (P[i] == c) { T[++i] = ++j; }
        else if (j > 0) { j = T[j];}
        else { T[++i] = 0; j = 0;}
        c = P[j];
    }
}
```

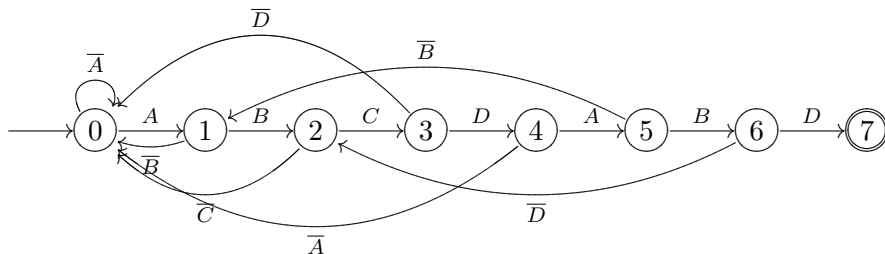
}

Algorithme de construction des correspondances partielles (en C)

```
void kmp_tableau(char *P)
{
    extern int T[];
    int i = 0;
    int j = -1;
    char c = '\0';
    T[0] = j; /* = -1 */
    while (P[i] != '\0')
    {
        if (P[i] == c) { T[++i] = ++j; }
        else if (j > 0) { j = T[j];}
        else { T[++i] = 0; j = 0;}
        c = P[j];
    }
}
```


Un automate

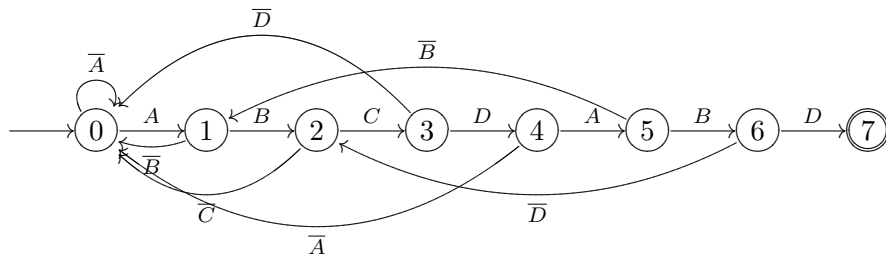
- Considérons l'automate suivant



- Cet automate trace les différentes valeurs de l'indice j dans l'algorithme précédent pour la chaîne ABCDABD. j peut soit être incrémenté, soit retourner à la valeur 0.

Un automate

- Considérons l'automate suivant



- Cet automate trace les différentes valeurs de l'indice j dans l'algorithme précédent pour la chaîne ABCDABD. j peut soit être incrémenté, soit retourner à la valeur 0.

Algorithme KPM (version automate) (en C)

```

int kmp_automate(char *P, char *S)
{
    extern int T[];
    int j = 0;
    for (int m = 0; S[m] != '\0'; ++m)
        for (;;)
        {
            if (S[m] == P[j])
            { ++j; if (P[j] == '\0'){ return m-j;}; break; }
            else
            { if (j == 0) break;
              else { j = T[j];}
            }
        }
}

```

Algorithme KPM (version automate) (en C)

```

int kmp_automate(char *P, char *S)
{
    extern int T[];
    int j = 0;
    for (int m = 0; S[m] != '\0'; ++m)
        for (;;)
        {
            if (S[m] == P[j])
            { ++j; if (P[j] == '\0'){ return m-j;}; break; }
            else
            { if (j == 0) break;
              else { j = T[j];}
            }
        }
}

```

Algorithme KPM (version automate) (en C)

```

int kmp_automate(char *P, char *S)
{
    extern int T[];
    int j = 0;
    for (int m = 0; S[m] != '\0'; ++m)
        for (;;)
        {
            if (S[m] == P[j])
            { ++j; if (P[j] == '\0'){ return m-j;}; break; }
            else
            { if (j == 0) break;
              else { j = T[j];}
            }
        }
}

```

Algorithme KPM (version automate)

Cet algorithme a plusieurs avantages par rapport au précédent :

- Les caractères de S sont testés un par un
- il n'y a donc pas d'accès aléatoire dans le tableau S
- Cette version de l'algorithme KPM peut être utilisée pour travailler avec un flux d'entrée plutôt qu'une chaîne stockée en mémoire.

Algorithme KPM (version automate)



Cet algorithme a plusieurs avantages par rapport au précédent :

- Les caractères de S sont testés un par un
- il n'y a donc pas d'accès aléatoire dans le tableau S
- Cette version de l'algorithme KPM peut être utilisée pour travailler avec un flux d'entrée plutôt qu'une chaîne stockée en mémoire.

Algorithme KPM (version automate)

Cet algorithme a plusieurs avantages par rapport au précédent :

- Les caractères de S sont testés un par un
- il n'y a donc pas d'accès aléatoire dans le tableau S
- Cette version de l'algorithme KPM peut être utilisée pour travailler avec un flux d'entrée plutôt qu'une chaîne stockée en mémoire.

-  Donald Knuth, James H. Morris, Jr. et Vaughan Pratt. *Fast pattern matching in strings*. SIAM Journal on Computing, 6(2):323–350. 1977.
-  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001. Section 32.4: The Knuth-Morris-Pratt algorithm, pp.923–931.

Liens

- <http://www.ics.uci.edu/~eppstein/161/960227.html> Une explication de l'algorithme.
- <http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html> Un exemple de l'algorithme Knuth-Morris-Pratt, sur la page Internet de J Strother Moore, co-inventeur de l'algorithme de Boyer-Moore.
- <http://www-igm.univ-mlv.fr/~lecroq/string/node8.html> Une page consacrée à l'algorithme, sur un site dévolu aux algorithmes de texte.