

INF280

Algorithmes de texte

Pierre Senellart, Antoine Amarilli

Table des matières

Recherche de sous-chaîne

Expressions rationnelles et automates

Algorithme de Huffman

Conclusion

Trouver les occurrences d'une sous-chaîne dans une chaîne

- Algorithme naïf :

```
// s est la chaîne, p le motif
```

```
for (int i=0, j; i < s.size() - p.size() + 1; ++i) {  
    for (j = 0; j < p.size() && s[i+j] == p[j]; ++j)  
        ;  
  
    if (j == p.size())  
        printf("Match à la position %d\n", i);  
}
```

- En général, similaire à l'**implémentation native** du langage (strstr, fortement optimisée)
- Complexité $O(|s| \times |p|)$
- Peut-on **mieux faire** ?

Knuth–Morris–Pratt : idée

- Motif p , chaîne s
- Pour chaque préfixe p' de p , maintenir la taille du préfixe maximal de p qui est un suffixe strict de p'
- $p = \text{"abcababcabd"}$
00012123450
- Tableau constructible en temps linéaire en le motif p

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T													
i													
cnd													

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1											
i													
cnd													

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T	-1												
i			↑										
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0										
i			↑										
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0										
i			↑										
cnd	↑												

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0										
i			↑										
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0										
i				↑									
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0									
i				↑									
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0									
i				↑									
cnd	↑												

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0									
i				↑									
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0									
i					↑								
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0								
i					↑								
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0								
i					↑								
cnd			↑										

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0								
i						↑							
cnd			↑										

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1							
i						↑							
cnd			↑										

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1							
i						↑							
cnd				↑									

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1							
i							↑						
cnd				↑									

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2						
i							↑						
cnd				↑									

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2						
i							↑						
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2						
i							↑						
cnd			↑										

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2						
i								↑					
cnd			↑										

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1					
i								↑					
cnd			↑										

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1					
i								↑					
cnd				↑									

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1					
i									↑				
cnd				↑									

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2				
i									↑				
cnd				↑									

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2				
i									↑				
cnd					↑								

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2				
i										↑			
cnd					↑								

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3			
i										↑			
cnd					↑								

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3			
i										↑			
cnd						↑							

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3			
i													
cnd						↑						↑	

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4		
i											↑		
cnd						↑							

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4		
i											↑		
cnd							↑						

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4		
i													↑
cnd							↑						

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	
i													↑
cnd							↑						

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	
i													↑
cnd				↑									

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	
i													↑
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	
i													↑
cnd	↑												

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	
i													↑
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	
i													↑
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	0
i													↑
cnd		↑											

Knuth–Morris–Pratt : calcul du tableau

```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	0
i													↑
cnd	↑												

Knuth–Morris–Pratt : calcul du tableau

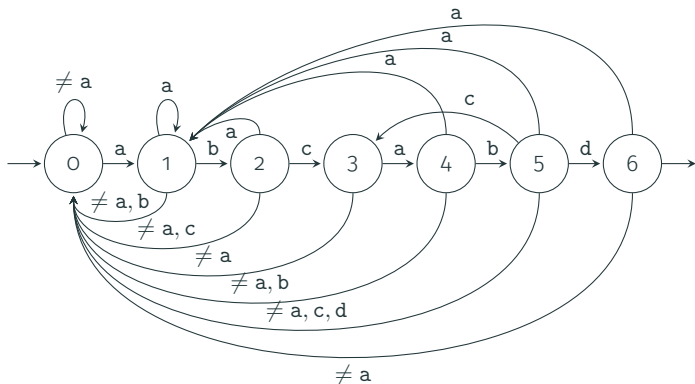
```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
    T[i] = cnd;
    while (cnd >= 0 && p[cnd] != p[i])
        cnd = T[cnd];
    cnd++;
}
```

	-1	0	1	2	3	4	5	6	7	8	9	10	11
p		a	b	c	a	b	a	b	c	a	b	d	
T		-1	0	0	0	1	2	1	2	3	4	5	0
i													↑
cnd		↑											

Knuth–Morris–Pratt : interprétation comme automate

T peut être vu comme l'**automate déterministe** des mots ayant pour suffixe p . Par exemple, pour $p = \text{abcabd}$:

	0	1	2	3	4	5	6
p	a	b	c	a	b	d	
T	-1	0	0	0	1	2	0



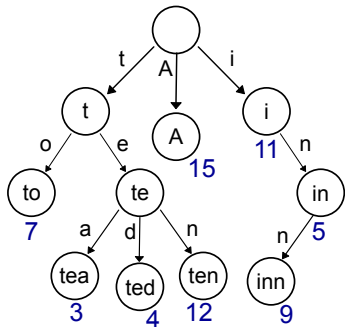
Knuth–Morris–Pratt: recherche

```
char p[MAXN]; int T[MAXN+1];
int np = strlen(p), ns = strlen(s);
[...] // ici, construire la table T comme précédemment
int cnd = 0; // position courante dans le motif p
for (int i = 0; i <= ns; i++) { // tant qu'on ne lit pas
    while (cnd >= 0 && p[cnd] != s[i]) // le prochain char de p
        cnd = T[cnd]; // on recule dans p
    cnd++; // maintenant que le prochain char convient, avancer
    if (cnd == np) {
        // on a atteint la fin de p, donc on a trouvé un match
        printf("match at %d\n", i - np + 1);
        // on recule dans p au cas où le prochain match chevauche
        cnd = T[cnd];
    }
}
```


Recherche d'un mot parmi un ensemble de mots

- **Dictionnaire** D de n mots
- Pour rechercher si un mot est dans cet ensemble :
 - **Vecteur** (vector) ou **liste chaînée** (forward_list) : $O(n)$
 - **Arbre binaire équilibré** (set) : $O(\log n)$
 - **Table de hachage** (unordered_set) : $O(1)$
mais $O(n)$ dans le pire cas (collisions)
 - **Trie** (ou arbre préfixe) : $O(1)$

Trie (arbre préfixe)

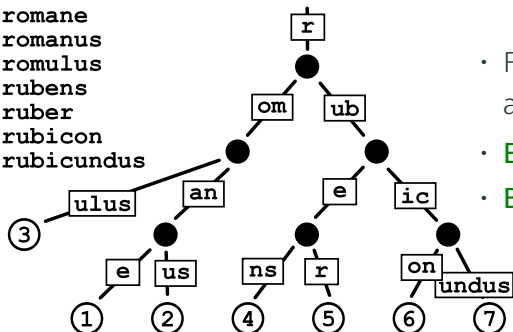


Trie_example.svg, Domaine public,
Chris-martin, Wikimedia Commons

- Arbre des **préfixes** des mots de D
- Arêtes **étiquetées** par des lettres
- On indique sur chaque nœud l'id dans D du **chemin** qui y mène
- Permet de trouver les **continuations** du mot d'entrée m dans D , i.e., les mots de D dont m est **préfixe**

Arbre radix (trie Patricia)

- 1 romane
- 2 romanus
- 3 romulus
- 4 rubens
- 5 ruber
- 6 rubicon
- 7 rubicundus



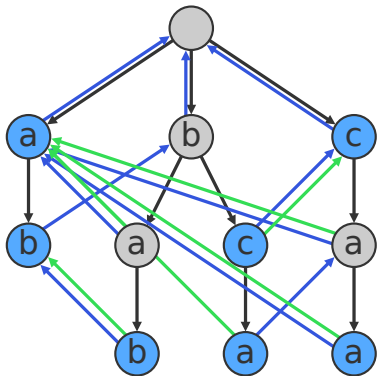
- Fusionne les **enfants uniques** avec leur parent
- **Espace mémoire** réduit
- **Branchement :**
 - par bit (entiers),
 - par lettre,
 - par groupe de bits (radix : taille du groupe)

Patricia trie.svg, CC-BY 2.5, Claudio Rocchini, Wikimedia Commons

Trouver les occurrences d'un ensemble de sous-chaînes dans une chaîne

- Dictionnaire D de taille n contenant des mots de taille k
- Chaîne de taille l
- Applications répétées de Knuth–Morris–Pratt ? $O(n \times (k + l))$
- On peut généraliser Knuth–Morris–Pratt à un trie arbitraire (et non une séquence de caractères) : $O(n \times k + l + m)$ où m est le nombre total d'occurrences (taille du résultat), au plus $n \times l$ mais possiblement plus faible

Aho-Corasick



Ahocorasick.svg, CC-BY-SA 3.0,
Dllu, Wikimedia Commons

Dictionnaire : a, ab, bab, bc,
bca, c, caa.

- Construire un **trie** du dictionnaire (liens en noir, mots sur fond **bleu**)
- Ajouter des **pointeurs** (en **bleu**) vers le plus grand suffixe strict dans le trie
- Ajouter des **raccourcis** (en **vert**) pour les mots du dictionnaire
- Exemple : **abccab** donne :
 - **a** (parcours normal),
 - **ab** (parcours normal),
 - **bc** (suivi du pointeur **bleu**) puis **c** (clôture par le pointeur **vert**),
 - **c** (suivi du pointeur **bleu**),
 - **ca** (parcours normal) puis **a** (clôture par le pointeur **vert**),
 - **ab** (suivi du pointeur **bleu**)

Aho-Corasick : Construction du trie

```
int nw; // nombre de mots du dictionnaire
char w[MAXW][MAXL]; // contenu des mots du dictionnaire
int trie[MAXW*MAXL+2][ALPHA]; // trie des mots du dictionnaire
int fs; // prochain index libre dans trie
int endw[MAXW*MAXL+2]; // mot qui se finit à cet index de trie

// insérer le mot numéro i de contenu s dans le trie
// en partant du nœud d'index n dans le trie
void insert_trie(int i, char *s, int n) {
    unsigned char x = s[0];
    if (!x) { endw[n] = i; return; } // fin du mot atteinte
    if (!trie[n][x]) // si pas d'arête étiquetée x depuis n...
        trie[n][x] = fs++; // ... on crée un nouveau nœud cible
    return insert_trie(i, s+1, trie[n][x]);
}
```

Aho-Corasick : Début du code

```
void aho_corasick() {  
    // l'index 0 dans le trie indique les nœuds inexistantes  
    // la racine du trie est 1  
    // donc initialement le prochain index libre est 2  
    fs = 2;  
    // les mots sont numérotés à partir de 1  
    // 0 dans endw indique qu'aucun mot ne se termine  
    for (int i = 1; i <= nw; i++)  
        insert_trie(i, w[i], 1); // insère chaque mot dans le trie  
  
    [...] // ici, on calcule les pointeurs et les raccourcis  
}
```

Aho-Corasick : Calcul des pointeurs et raccourcis (1/2)

```
int pointer[MAXW*MAXL+2]; // pointeurs (en bleu), cf T de KMP
int shortcut[MAXW*MAXL+2]; // raccourcis (en vert)
queue<int> q;
q.push(1); // explorer en BFS depuis la racine du trie
while (!q.empty()) {
    int n = q.front();
    q.pop();
    for (unsigned char x = 0; x < ALPHA; x++) {
        int n2 = trie[n][x];
        if (!n2)
            continue; // pas d'arête pour cette lettre depuis n
        [...] // on traite n2, cf prochain transparent
        q.push(n2); // continuer le BFS avec n2
    }
}
```


Aho-Corasick : Calcul des pointeurs et raccourcis (2/2)

```
// nœud n2 de parent n avec étiquette x, i.e., n -x-> n2
// == calcul du pointeur ==
int p = pointer[n];           // p est le pointeur de n
while (p && !trie[p][x])     // tant que p n'a pas d'arête x...
    p = pointer[p];         // ... on recule p via le pointeur
if (!p)                      // si p = 0, on est sorti du trie...
    pointer[n2] = 1;         // ... donc on met pointeur = racine
else                          // sinon le pointeur est...
    pointer[n2] = trie[p][x]; // ... le x-successeur de p

// == calcul du raccourci ==
if (endw[pointer[n2]])       // si un mot finit en n2...
    shortcut[n2] = pointer[n2]; // ... raccourci = pointeur
else                          // sinon le raccourci est celui du pointeur
    shortcut[n2] = shortcut[pointer[n2]];
```

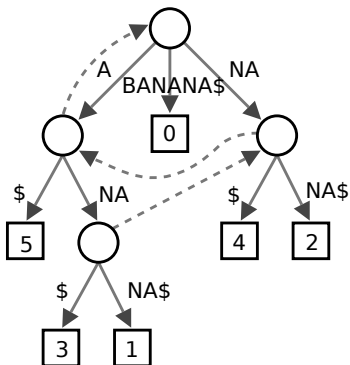
Aho-Corasick : Utilisation

```
int pos = 1; // position courante dans le trie
char s[MAXS]; int ns = strlen(s); // chaîne à parcourir

for (int i = 0; i < ns; i++) {
    unsigned char x = s[i]; // x = nouveau caractère lu
    while (pos && !trie[pos][x]) // tant que pas d'arête x depuis p...
        pos = pointer[pos]; // ... on recule pos suivant pointeur
    pos = trie[pos][x]; // maintenant on essaie d'avancer par x
    if (!pos) // si on est sorti du trie...
        pos = 1; // ... on revient à la racine

    int posb = pos; // on va énumérer les matches possibles depuis pos...
    do { // fin de mot possible en i
        if (endw[posb]) // on a vraiment une fin de mot à posb (en i):
            printf("match of %s at %d\n", // afficher
                w[endw[posb]], i - strlen(endw[posb]) + 1); // le match
        posb = shortcut[posb]; // suivre le raccourci de posb
    } while (endw[posb]); // ... tant que des mots se terminent
}
```

Arbre des suffixes



- Trie Patricia des **suffixes** d'un mot (ici, BANANA, suivi d'un délimiteur \$)
- Constructible en $O(n^2)$ de droite à gauche
- Constructible en $O(n)$ de gauche à droite, comme les pointeurs d'Aho-Corasick
- Permet d'**indexer** une chaîne pour rechercher des sous-chaînes
- Nombreuses **autres applications** : p. ex., plus longue sous-chaîne commune

Suffix tree BANANA.svg, Domaine public,
Maciej Jaros and Nils Grimsmo,
Wikimedia Commons

Table des matières

Recherche de sous-chaîne

Expressions rationnelles et automates

Algorithme de Huffman

Conclusion

Expressions rationnelles

- Langage permettant de décrire des **motifs** à rechercher dans une chaîne de caractères
- Par exemple : $(a|b)^*\#(a|b)^*(\#(a|b|\#)^*)?$
- Généralise la recherche de **sous-chaînes**, de mots d'un **dictionnaire**, de **préfixes**, de **suffixes**, etc.
- Processeurs d'expressions rationnelles dans la **bibliothèque standard** de C++ 2011 (`std::regex`)

- **Expression rationnelle** vers **automate fini nondéterministe** :
 - **algorithme de Thompson** : temps linéaire, transitions spontanées
 - **algorithme de Glushkov** : temps quadratique, moins d'états
- Reconnaître si une chaîne de longueur n est **acceptée** par un automate nondéterministe à m états est en $O(n \times m)$
- Un automate nondéterministe à m états peut être **transformé** en automate déterministe à $O(2^m)$ états en temps $O(2^m)$
- Reconnaître si une chaîne de longueur n est **acceptée** par un automate déterministe à m états est en $O(n)$

- Les “expressions rationnelles” de C++ incluent des **extensions** :
 - **Références arrières** (indiquer qu’une sous-chaîne se répète)
 - Opérateur * **glouton** et *? **réticent**
- Du coup, les implémentations des expressions rationnelles n’utilisent pas des automates, mais du **backtracking**
- Souvent **beaucoup moins efficaces** ! Exponentiel en la taille de l’expression rationnelle dans les cas pathologiques

Table des matières

Recherche de sous-chaîne

Expressions rationnelles et automates

Algorithme de Huffman

Conclusion

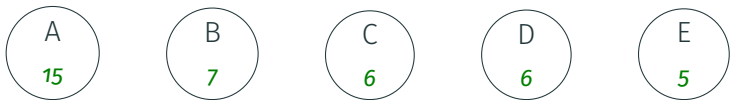
Algorithme de Huffman : Objectif

- Ensemble de **symboles** s_i avec un **poids** w_i (e.g., probabilité)
- On veut choisir un **code** C_i (mot sur $\{0, 1\}$) pour chaque mot
- **Code préfixe** : aucun mot code n'est préfixe d'un autre
- Minimiser le **poids moyen** : $\sum_i |C_i| \times w_i$
 - **Intuition** : un mot **lourd** (fréquent), doit avoir un code **bref**

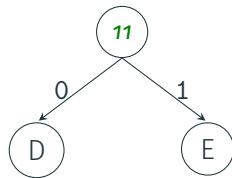
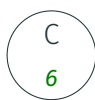
Algorithme de Huffman : Principe

- Représenter un **code préfixe** comme un **arbre de décision**
- Symboles *a, b, c, d, e*

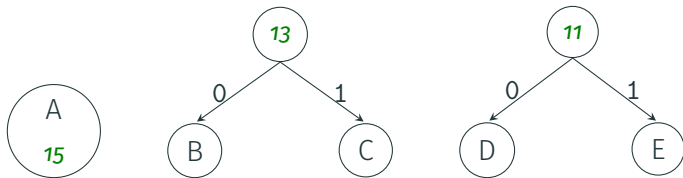
Algorithme de Huffman : Exemple



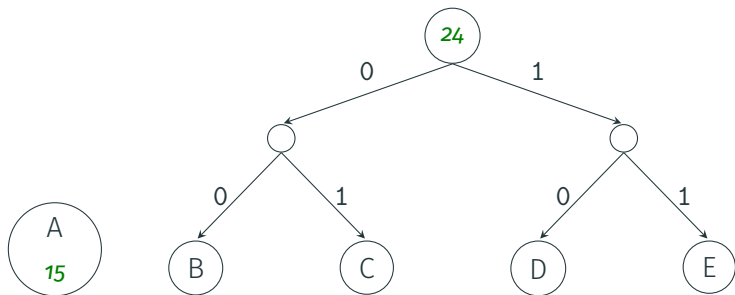
Algorithme de Huffman : Exemple



Algorithme de Huffman : Exemple



Algorithme de Huffman : Exemple



Algorithme de Huffman : Exemple

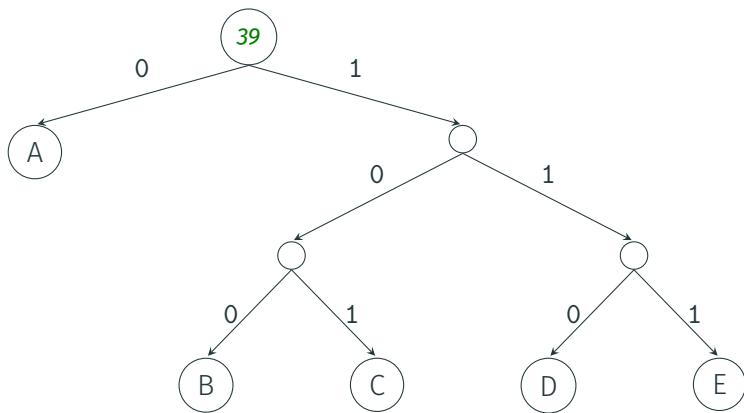


Table des matières

Recherche de sous-chaîne

Expressions rationnelles et automates

Algorithme de Huffman

Conclusion

- Déterminer si **une chaîne est dans un ensemble de chaînes** :
 - table de hachage
- Trouver les **chaînes du dictionnaire dont une chaîne est préfixe** :
 - trie, arbre radix
- Recherche d'une **petite sous-chaîne dans une chaîne** :
 - implémentation native du langage de programmation
- Recherche d'une **longue sous-chaîne dans une longue chaîne** :
 - Knuth–Morris–Pratt (ou Boyer–Moore, non traité)
- Recherche d'un **dictionnaire de sous-chaînes dans une chaîne** :
 - Aho–Corasick (indexe les sous-chaînes)
 - arbre des suffixes (indexe la chaîne)
- Recherche d'un **motif complexe dans une chaîne** :
 - expression rationnelle ou automate compilé à partir du motif