Algorithmes de texte

Cours L3 Algorithmique et programmation

Pierre Senellart, Antoine Amarilli

23 novembre 2017

Plan

Recherche dans une chaîne

Recherche dans un ensemble de chaînes

Recherche d'ensemble de chaînes dans une chaîne

Expressions rationnelles

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`, `string::find`), fortement optimisée

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`, `string::find`), 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'
- $p =$“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

```c
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

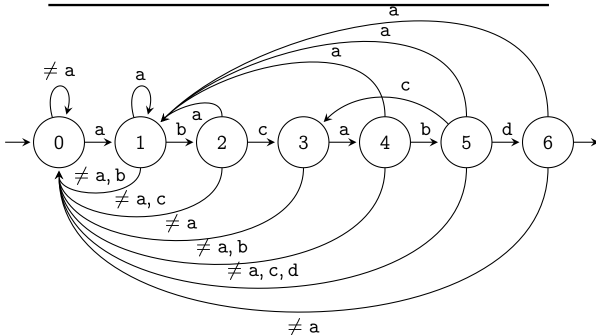
```
char p[MAXN]; int T[MAXN+1]; int np = strlen(p);
T[0] = -1;
int cnd = 0;
for (int i = 1; i <= np; i++) {
  T[i] = cnd;
  while (cnd >= 0 && p[cnd] != p[i])
    cnd = T[cnd];
  cnd++;
}
```

	-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];
  }
}
```

Plan

Recherche dans une chaîne

Recherche dans un ensemble de chaînes

Recherche d'ensemble de chaînes dans une chaîne

Expressions rationnelles

Conclusion

Recherche d'un mot parmi un ensemble de mots

- Dictionnaire D de n mots de longueur $\leqslant \ell$
- Pour rechercher si un mot est dans cet ensemble :
  - Tableau (`vector`) ou liste chaînée (`forward_list`) : $O(n \times \ell)$
  - Arbre binaire équilibré (`set`) : $O(\log n \times \ell)$
  - Table de hachage (`unordered_set`): $O(\ell)$ mais $O(n \times \ell)$ dans le pire cas (collisions)

Recherche d'un mot parmi un ensemble de mots

- Dictionnaire D de n mots de longueur $\leqslant \ell$
- Pour rechercher si un mot est dans cet ensemble :
  - Tableau (`vector`) ou liste chaînée (`forward_list`) : $O(n \times \ell)$
  - Arbre binaire équilibré (`set`) : $O(\log n \times \ell)$
  - Table de hachage (`unordered_set`): $O(\ell)$ mais $O(n \times \ell)$ dans le pire cas (collisions)
  - Filtre de Bloom : $O(\ell)$ et occupation mémoire indépendante de ℓ, mais possibilité de faux positifs

Recherche d'un mot parmi un ensemble de mots

- Dictionnaire D de n mots de longueur $\leqslant \ell$
- Pour rechercher si un mot est dans cet ensemble :
    - Tableau (`vector`) ou liste chaînée (`forward_list`) : $O(n \times \ell)$
    - Arbre binaire équilibré (`set`) : $O(\log n \times \ell)$
    - Table de hachage (`unordered_set`): $O(\ell)$ mais $O(n \times \ell)$ dans le pire cas (collisions)
    - Filtre de Bloom : $O(\ell)$ et occupation mémoire indépendante de ℓ, mais possibilité de faux positifs
    - Trie (ou arbre préfixe) : $O(\ell)$ et occupation mémoire plus faible que structures de données classiques

Filtre de Bloom

Au tableau.

Filtre de Bloom

Au tableau.
Antisèche :

- tableau de taille m, n éléments, k fonctions de hachage
- Probabilité d'erreur:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \sim \left(1 - \mathrm{e}^{-kn/m}\right)^k$$

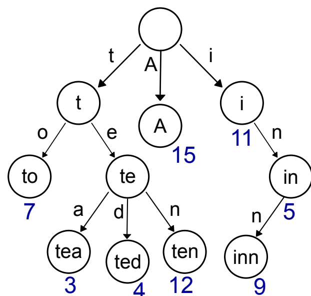
- k optimal:

$$\frac{m \ln 2}{n}$$

- $\frac{m}{n}$ optimal:

$$-\frac{\log_2 p}{\ln 2}$$

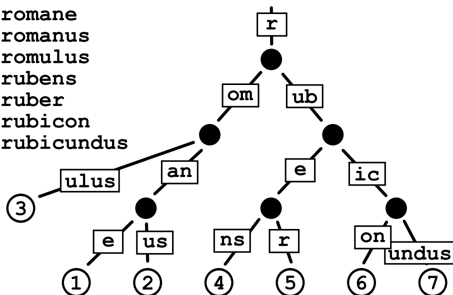
Trie (arbre préfixe)



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
- L'ordre lexicographique est préservé
- Bonne occupation mémoire si l'alphabet est petit

Arbre radix (trie Patricia)



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

Wikimedia Commons

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

Plan

Recherche dans une chaîne

Recherche dans un ensemble de chaînes

Recherche d'ensemble de chaînes dans une chaîne

Expressions rationnelles

Conclusion

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

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 ?
  $O(n \times (k + \ell))$

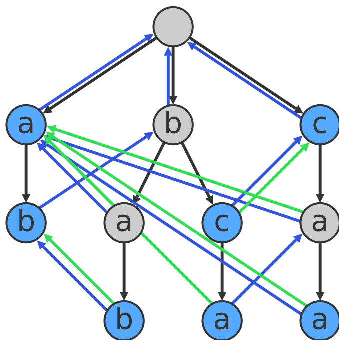
Trouver les occurrences d'un ensemble de sous-chaînes dans une chaîne

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

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 ?
  $O(n \times (k + \ell))$
- On peut généraliser Knuth–Morris–Pratt à un trie arbitraire (et non une séquence de caractères) :
  $O(n \times k + \ell + m)$ où m est le nombre total d'occurrences (taille du résultat), au plus $k \times \ell$ mais possiblement plus faible
- Approche alternative : indexer la chaîne plutôt qu'indexer les sous-chaînes $\Rightarrow$ arbre des suffixes, même complexité

Aho–Corasick

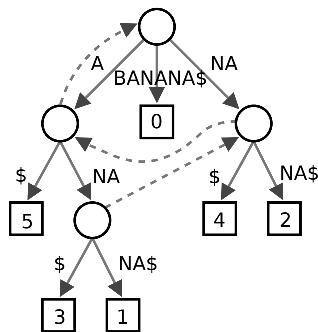


Ahocorasick.svg, CC-BY-SA 3.0,
Dllu, Wikimedia Commons

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 (suivi du pointeur bleu) puis a (clôture par le pointeur vert),
  - ab (suivi du pointeur bleu)

Arbre des suffixes



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

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

Plan

Recherche dans une chaîne

Recherche dans un ensemble de chaînes

Recherche d'ensemble de chaînes dans une chaîne

Expressions rationnelles

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 Python, Java, C++ 2011 (`std::regex`)...

Automates

- 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 $O(n)$

Expressions rationnelles et expressions rationnelles

- Les « expressions rationnelles » des langages de programmation 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

Plan

Recherche dans une chaîne

Recherche dans un ensemble de chaînes

Recherche d'ensemble de chaînes dans une chaîne

Expressions rationnelles

Conclusion

En résumé

- Déterminer si une chaîne est dans un ensemble de chaînes :
  → table de hachage, filtre de Bloom si faux positifs ok
- 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