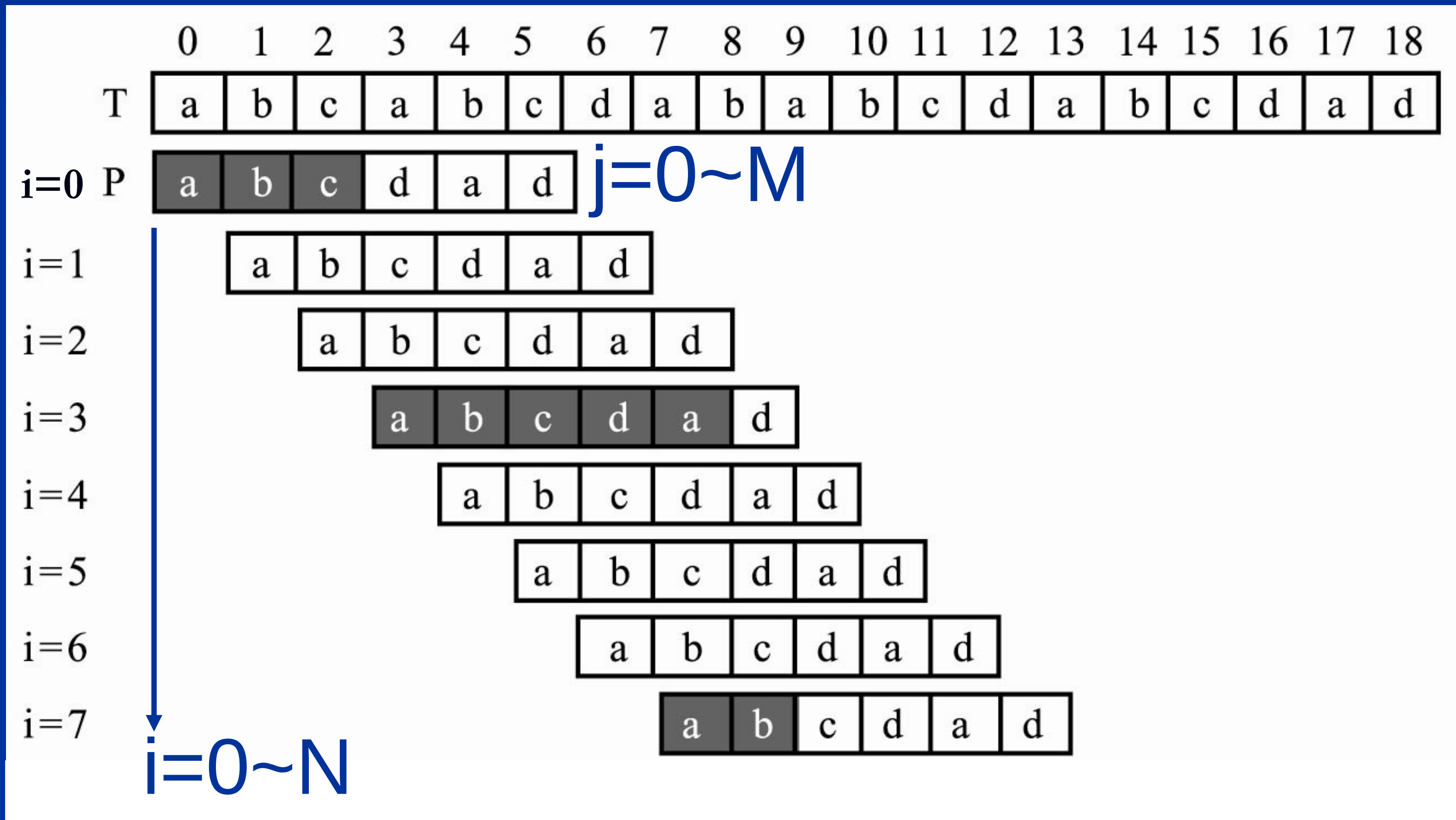
演算法 搜尋(字串)

- 暴力法
- Knuth-Morris-Pratt 演算法
- Boyer-Moore 演算法
- Rabin-Karp 演算法

暴力法搜尋字串

搜尋 T 字串(長度 N)是否含有 P 字串 (長度 M)



範例 8-1

暴力法搜尋字串

```
int brute_search(char* T, char* P){
   int i, j, N=strlen(T), M=strlen(P);
   for (i=0, j=0; j<M && i<N; )
        if ( T[i]==P[j] ) { i++; j++; }
          else { i=i-j+1; j=0; }
   if (j==M) return i-M; //found
     else return -1;        //fail
}
```

暴力法搜尋字串

範例 8-1

由標準輸入讀入

```
main(){
    int k, s, N=10000;  char T[N], P[]="algorithm";
    for(k=0; k<N; k++){
        T[k]=getchar(); if( T[k] == EOF ) break;
    } T[k-1]=0;
    for(k=0;k<5000;k++) s=brute_search(T, P);
    if (s!=-1)
        for(k=0; k<strlen(P); k++)
            printf("%c", T[s+k]);
}
```

搜尋字串

效能測試

執行結果

this is algorithm test.

algorithm

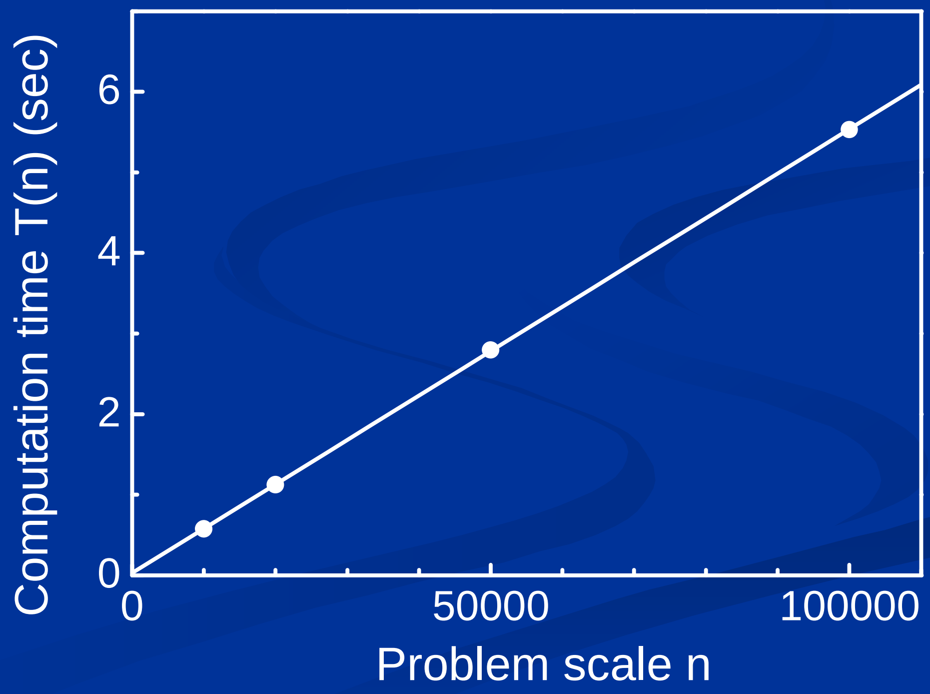
執行 5000次

n=10000 0.577s

n=20000 1.124s

n=50000 2.796s

n=100000 5.530s



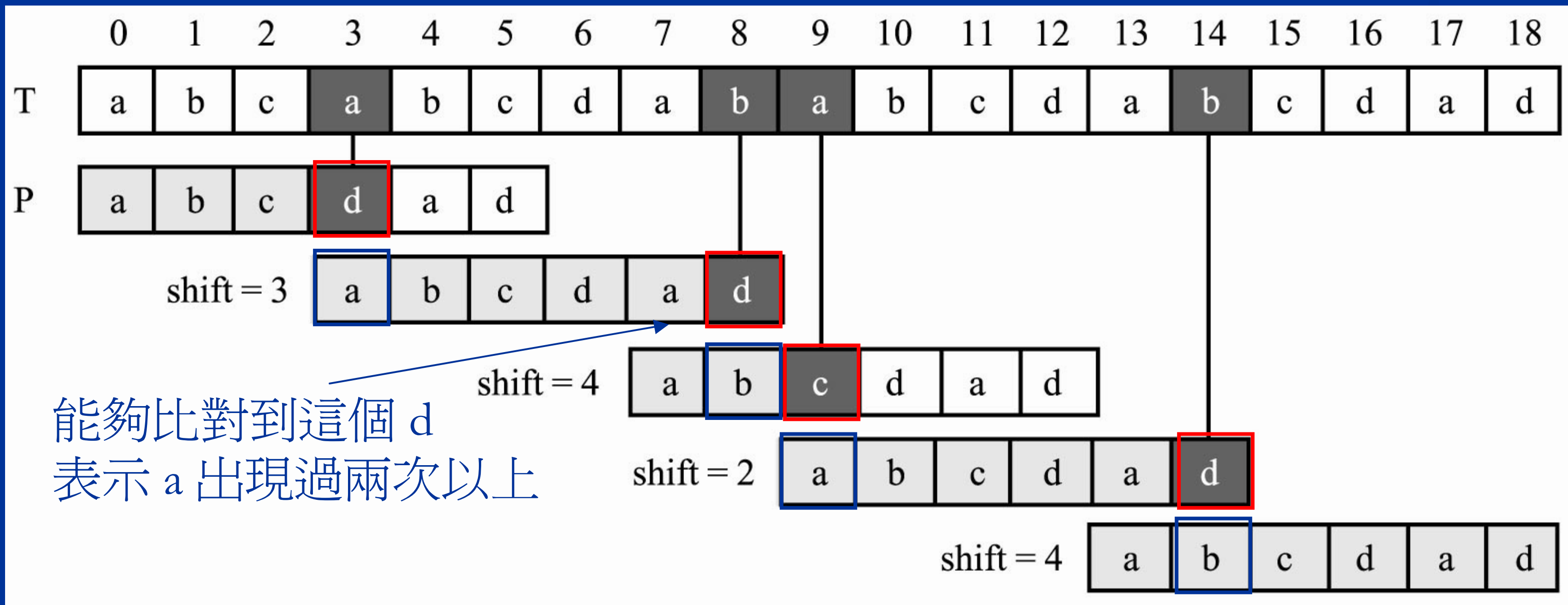
效能分析

- 成功：
  最糟每次比較 M 個，進行 N-M+1 次才成功
  假設 M << N，最糟時間複雜度 $\Theta(MN)$
  但 P 字串通常只比開頭字, 平均時間複雜度爲 $\Theta(N)$

  失敗：比成功最糟情況多一次
  最糟時間複雜度 $\Theta(MN)$
  平均時間複雜度 $\Theta(N)$

Knuth-Morris-Pratt 演算法

- 當字元比對不符時，已知字串 P 中某些字元可以略過不檢查, 直接跳到文件 T 下一個需要檢查的字元



失誤函數

- 先建構一個部分相符表格(失誤函數) next[M]

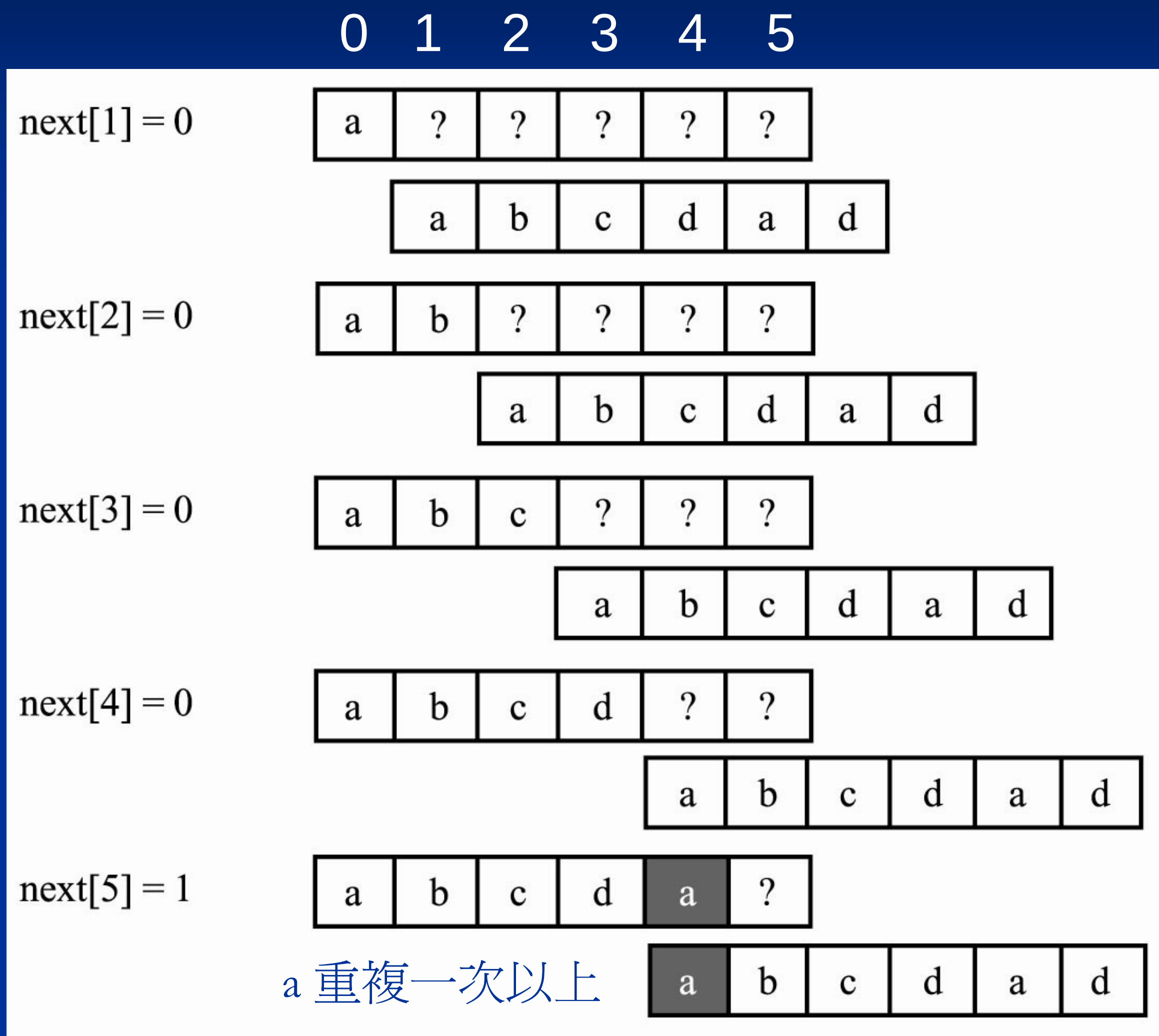
- 比對失敗發生在 P[j] 時
  以 P[next[j]] 作爲下一個重新比對的起點
  P={ a, b, c, d, a, d }
  next={ -1, 0, 0, 0, 0, 1 }
  -1 表示連 a 都不符合, 強迫前進
  0 表示可能符合前面的字元, 需要從 a 比較
  大於 0 表示符合前面的字元一次以上

範例 8-2

KMP 搜尋字串

```
int kmp_search( char *T, char* P){
   int i, j, N=strlen(T), M=strlen(P), next[M];
   create_next(P, next);
   for ( i=0, j=0; j<M && i<N; i++, j++)
        while( (j>=0) && (T[i]!=P[j]) ) j=next[j];
   if( j==M) return (i-M);
   else return -1;
}
```

失誤函數(failure function)



失誤函數

```
int create_next(char * P, int * next){
   int i,j;
   int M=strlen(P);
   next[0]= -1;
   for( i=0, j=-1; i<M; i++, j++, next[i]=j)
        while((j>=0) && (P[i]!=P[j])) j=next[j];
}
```

範例 8-2

KMP 搜尋字串

由標準輸入讀入

```
main(){
    int k, s, N=10000;  char T[N], P[]="algorithm";
    for(k=0; k<N; k++){
        T[k]=getchar(); if( T[k] == EOF ) break;
    } T[k-1]=0;
    for(k=0;k<5000;k++) s=kmp_search(T, P);
    if (s!=-1)
        for(k=0; k<strlen(P); k++)
            printf("%c", T[s+k]);
}
```

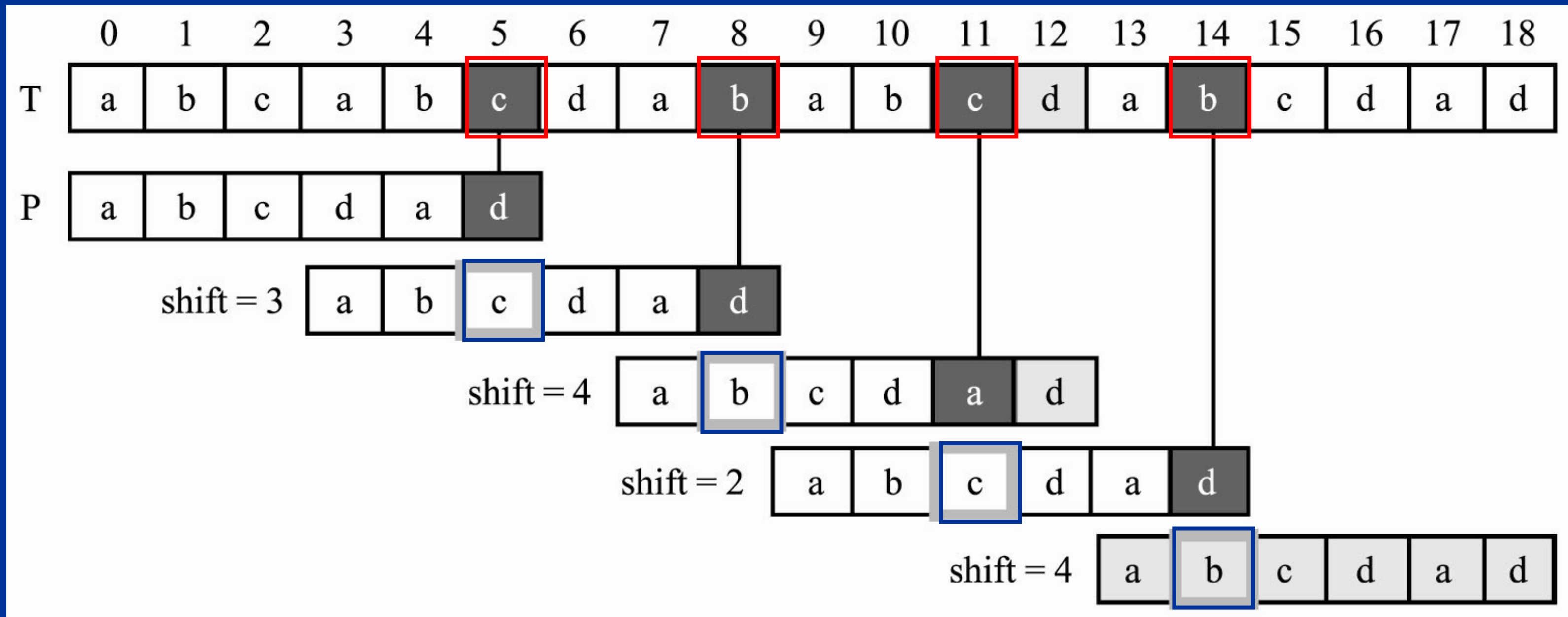
搜尋字串

KMP 演算法分析

- 建構 next 表格的複雜度為 Θ(M)
  平均時間複雜度 Θ(M+N), 最糟 Θ(NM)
- T 字串的索引 i 不會遞減
  適用於搜尋存放在硬碟的大型文件
- KMP 法比上暴力法增加效能有限
  只在 T 字串重複 P 部分多時較快
  平均時間通常仍花費在比對第一個字元上

Horspool algorithm

- 從字串 P 後面開始比對
  如果字串 P 的最後一個字元不符合，且這段 T 之間任一個字元不出現在 P 中，直接從下一段開始



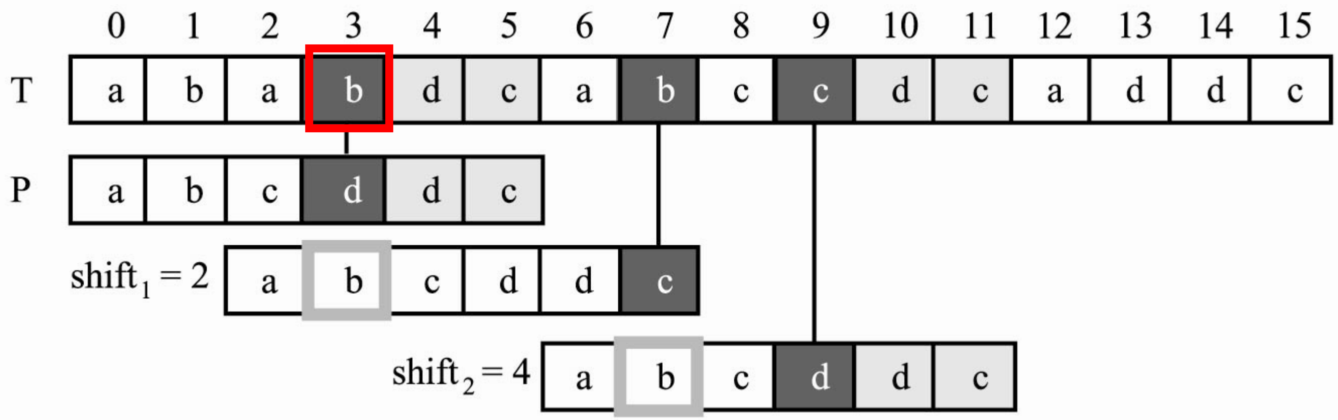
- 最佳只需 N/M 回合, 平均 Θ(M+N), 最糟 Θ(MN)

skip 表格

- 當T[i] 比對失敗時，用來計算位移大小
  →不相符字元位移

skip

character	a	b	c	d	other
shift	5	4	0	1	6



$i_{next} = 3 + skip$ [“b”] $= 7$, $shift_1 = 2$

$i_{next} = 7 + skip$ [“b”] $= 11$, $shift_2 = 4$

$i_{next} = 9 + skip$ [“c”] $= 9$, $shift_3 = 0$ ←無法繼續前進!!

不相符字元位移

```
#define MAX_KEY 256
#define index(x) (x)
int skip[MAX_KEY];
int create_skip(char *P){
   int i, M=strlen(P);
   for (i=0; i<MAX_KEY; i++) skip[i]=M;
   for (i=0; i<M; i++) skip[index(P[i])]=M-i-1;
}
```

範例 8-3

Horspool 搜尋字串

```
int horspool_search(char *T, char *P){
   int i, j, k, N=strlen(T), M=strlen(P);
   create_skip(P);
   for (i=M-1, j=M-1; j>0; i--, j--)
        while (T[i] != P[j]) {
                k=skip[index(T[i])];
                i += (M-j > k)? (M-j): k ;
                if (i>=N) return -1;
                j=M-1;
        }
   return i;
}
```

強迫前進

效能測試

執行結果

this is algorithm test.

algorithm

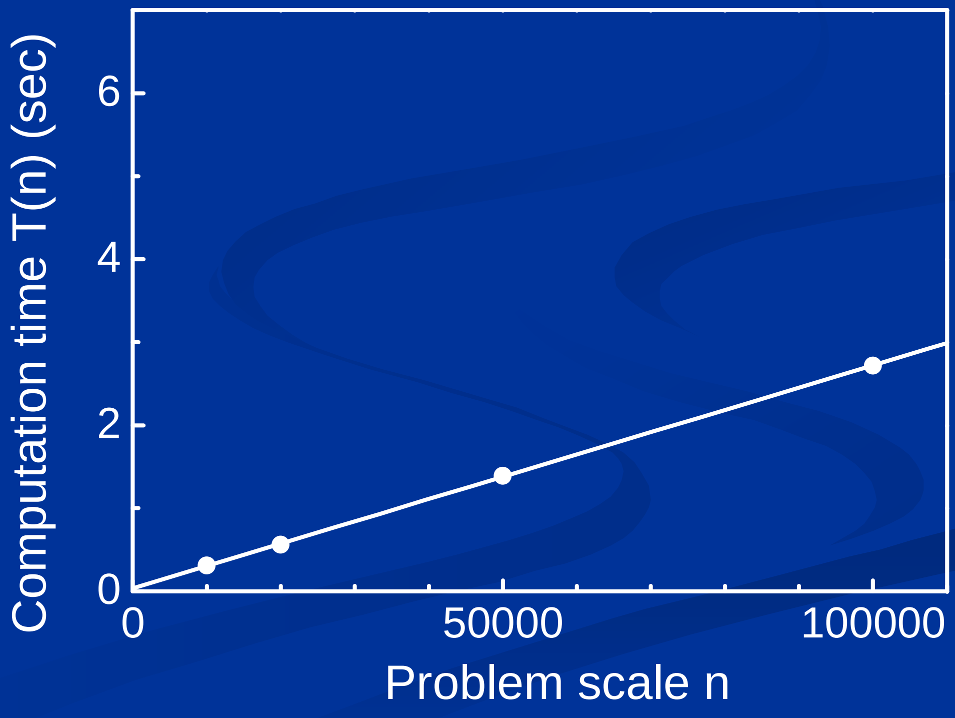
執行 5000次

n=10000 0.311s

n=20000 0.561s

n=50000 1.390s

n=100000 2.718s



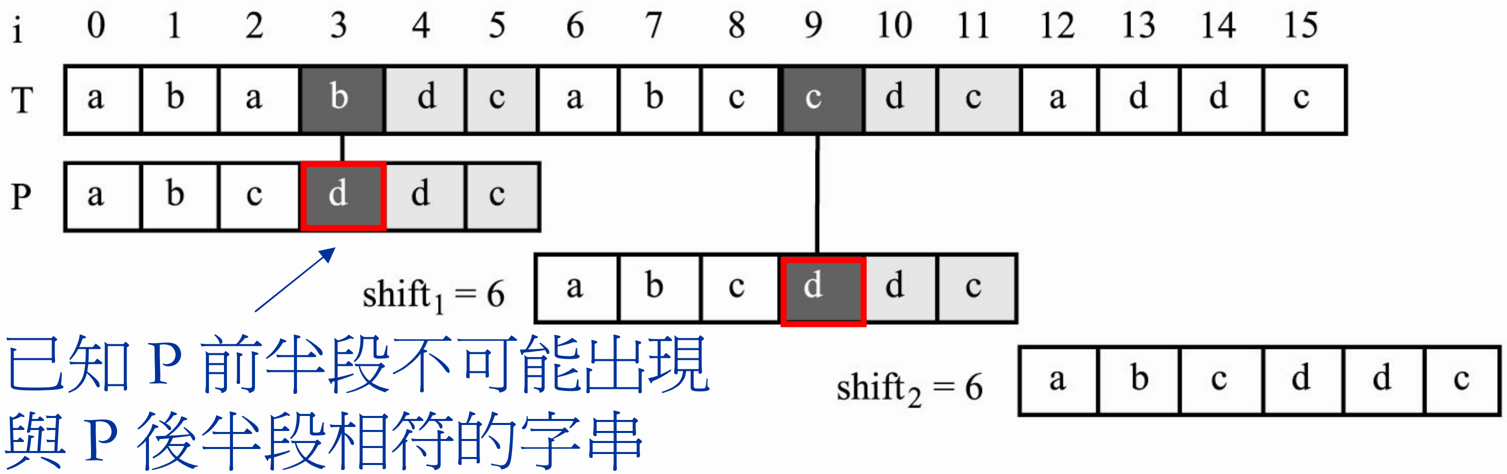
Boyer-Moore演算法

- 使用相符字尾位移

next

j	0	1	2	3	4	5
character	a	b	c	d	d	c
shift	6	6	6	6	3	1

(a)



$i_{next} = 5 + next[3] = 11$, $shift_1 = 6$

$i_{next} = 11 + next[3] = 17$, $shift_2 = 6$

(b)

BM 搜尋字串

範例 8-4

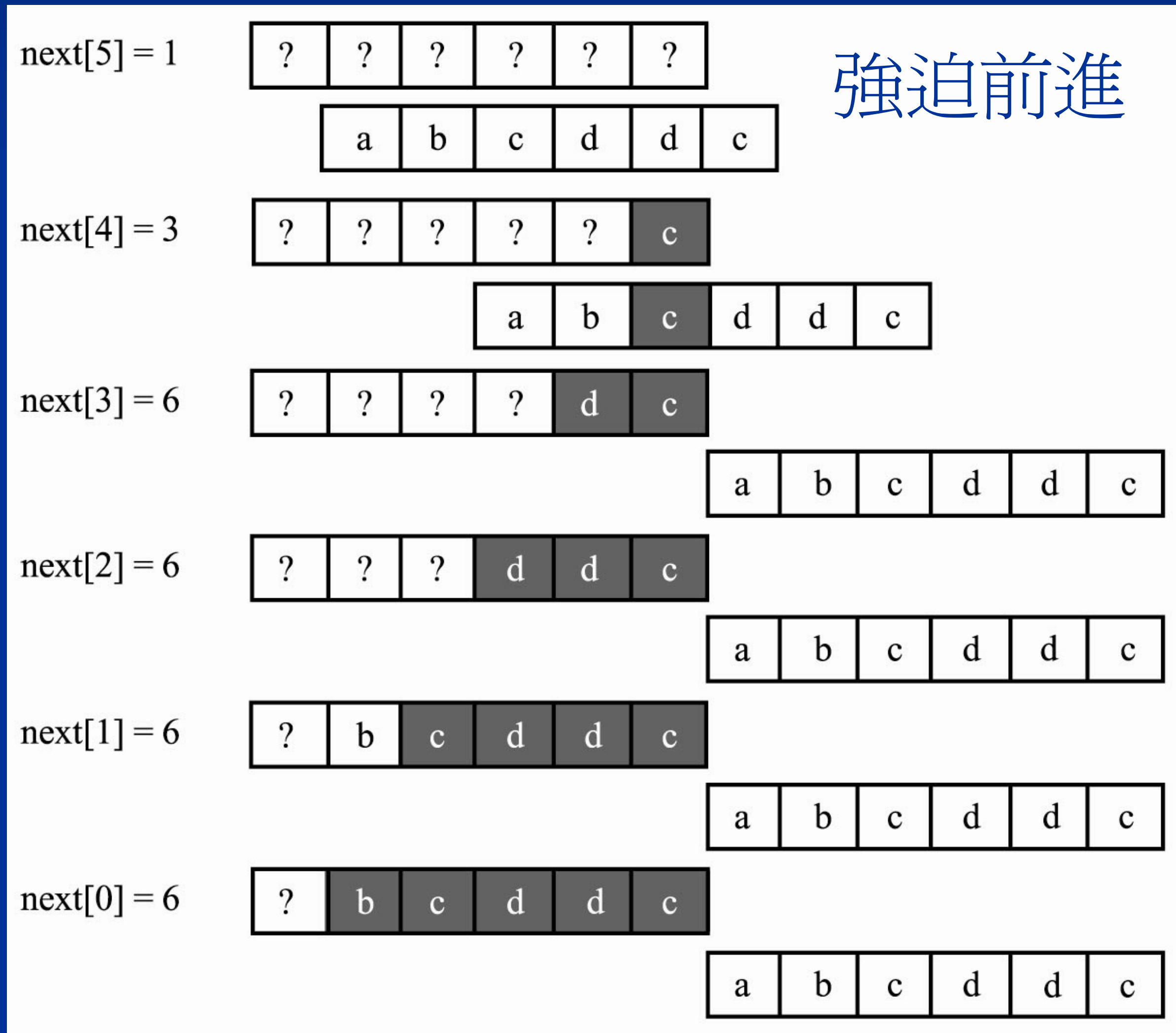
```
int bm_search(char *T, char *P){
   int i, j, k, k2, N=strlen(T), M=strlen(P); int next[M];
   create_skip(P);          create_bm_next(P, next);
   for (i=M-1, j=M-1; j>0; i--, j--)
        while (T[i] != P[j]) {
              k=skip[index(T[i])];     k2=next[j];
                i = (i+k > i-j+M-1+k2) ? i+k : i-j+M-1+k2;
              if (i>=N) return -1;
              j=M-1;
        }
   return i;
}
```

不相符字元位移

相符字尾位移

Boyer-Moore演算法

- 相符字尾位移: 根據 T 與P相符部分作為位移



範例 8-4

相符字尾位移

```
int create_bm_next(char *P, int *next){
    int i, j, M=strlen(P);
    next[M-1]=1;
    for(i=M-1, j=M-2; j>=0; i--, next[i]=M-1-j)
        while(P[i]!=P[j]) j--;
    for( ;i>=0;i--) next[i]=M;
}
```

BM演算法分析

- 不相符字元位移和相符字尾位移各自有最壞情況
→同時使用這兩個表格，並取其較大的位移量
- 平均、最糟時間複雜度為 Θ(N)
(最糟須進行 3N 個比較)
加上建構表格，時間複雜度為 Θ(M+N)

Rabin-Karp演算法

- 透過雜湊函數將字串 P 和文件 T 的部分字串轉換成數字，直接比對雜湊值

  平均時間複雜度 Θ(M+N)
  最糟效能 Θ(NM), 連續雜湊函數碰撞

- 可同時搜尋多組字串(但限定長度相同)

Rabin-Karp演算法

- 將字串視爲數學上的 d 進位系統, d 爲字元範圍
  將 T[i]…T[i+M-1] 轉換成初始雜湊值：

$$x = T[i]d^{M-1} + T[i+1]d^{M-2} + .. + T[i+M-2]d + T[i+M-1]$$

- 字串向右位移一步，轉換下一個雜湊值：

$$x = (x - T[i]d^{M-1})d + T[i+M]$$

- 使用除法雜湊函數 h(x) = x mod q 防止整數溢位

Rabin-Karp演算法

q=33554393 d=32

P

a	l	g	o	r	i	t	h	m

h1=16627079

T

t	h	i	s		i	s		a	l	g	o	r	i	t	h	m	

t	h	i	s		i	s		a

h2=21873289

h	i	s		i	s		a	l

h2=16269085

h2=16627079

a	l	g	o	r	i	t	h	m

防止整數溢位

$A \bmod Q = a \qquad\qquad B \bmod Q = b$

a) $(A + B) \bmod Q = (a + b) \bmod Q$

b) $(AB) \bmod Q = (ab) \bmod Q$

c) if $A \geq B$ $\quad (A - B) \bmod Q$

$$= \begin{cases} (a - b + Q) \bmod Q & \text{if } a < b \\ (a - b) \bmod Q & \text{if } a \geq b \end{cases}$$

d) if $d \leq Q$

$$d^M \bmod Q = (d \cdot d^{M-1}) \bmod Q$$

$$= (d \cdot (d^{M-1} \bmod Q)) \bmod Q$$

範例 8-5

```
#define index(x) (x-'a')
int q=33554393, d=32;

int rk_search(char *T, char *P) {
   int i, j, dM=1, h1=0, h2=0, match=0;
   int N=strlen(T), M=strlen(P);
```

計算初始雜湊值

```
   for (i=1; i<M; i++) dM=(d*dM) % q;
   for (i=0; i<M; i++){
        h1=( h1*d+index(P[i]) ) % q;
        h2=( h2*d+index(T[i]) ) % q;
   }
```

(接下頁)

(接上頁) 範例 8-5

```
        for (i=0; h1!=h2 && (match == 0) ; i++) {
            if (h1 == h2){
                for (j=0; j<M; j++)
                        if (P[j] != T[i+j]) break;
                if (j==M) match = 1;
            }                   // 雜湊值相同, 才進行比對

#define MINUS_MOD(a,b,Q) ( (a>=b) ? (a-b)%Q : (a-b+Q)%Q )
            h2 = ( MINUS_MOD(h2, (index(T[i])*dM) % q, q) *d
                        + index(T[i+M]) ) % q;
            if (i > N-M) return -1;
        }
        return i;
}
```

計算下一個雜湊值