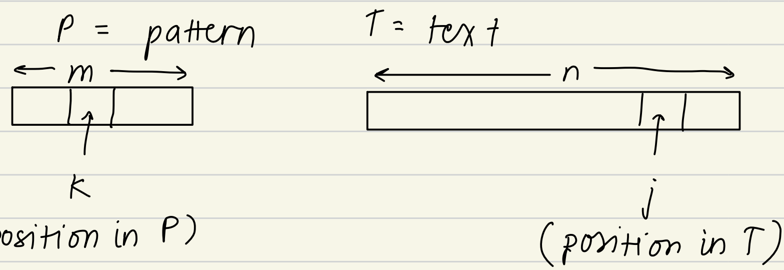
String Matching

problem: given a text T w/ n char
P w/ m char
find the first occurance of P in T

solutions
① straight forward
② Rabin - karp
③ Boyer - Moore

convections :



① Straight forward.

```
int simpleScan (char [] P, char [] T, int m) {
    int i, j, k ;  ← all
        ↳ i: guess of index of P in T
          j: position in T
          k: position in P
    while (j < n)
        if (T[j] == P[k]) {
            j++;
            k++;
            if (k == m) return i; }
        else
            j = ++i;
            k = 0;
            if (j > n-m) break; }
    return -1;
```

if mismatch just shift one to the right.

less than pattern left ⇒ not possible, so leave.

$n-m < j$ ← length check

P k m   i is index

T j n

$\rightarrow$ O(mn)

① Straight forward.

```
int simpleScan (char [] P, char [] T, int m){
    int i, j, k;  ← all
        ↳ i: guess of index of P in T
          j: position in T
          k: position in P
    while (j < n)
        if (T[j] == P[k]) {
            j++;
            k++;                      → j-k
            if (k == m) return i; }
      else         → j-k+1
        j = ++i;
        k = 0;
        if (j > n-m) break; }   return -1;
}
```

if mismatch just shift one to the right.

less than pattern left ⇒ not possible, so leave.

- worst case

  P : c . . . . . k   c for m-1

  k for m-1 th

  T : c c   c for all

  for every j, compare m times (mth comp breaks, move on to next j)

  j ↓ n

  $\therefore m \times (n-m+1)$ → last j is at n-m

  ∴ O(mn) comparisons.

- best case : T = P etc.   O(m) comparisons

$j: 1 \to m-1$ $p = p * d + P[j]$

$$t_j = (t_{j-1} - T[j] \cdot d^{n-1})\, d + T[j+m-1]$$

② Rabin-Karp Algorithm

① convert pattern (m char) to number : p
② convert first m char of text to number : t
③ if $p = t \to$ pattern found, exit
④ else if (! end of text)
shift window to the right by 1 and recompute ($n-m+1$)

for pattern of m characters

d = number of possible characters

a	b	c	.....	g	h	i

$\longleftarrow m \longrightarrow$

$$p = P[0] \cdot d^{m-1} + P[1] \cdot d^{m-2} + P[2] \cdot d^{m-3} + \vdots + P[m-1]$$

ⓐ
```
p = P[0] ;
for j = 1 to m-1
    p = p*d + P[j];
```
$\leftarrow \theta(m)$ for computation

ⓑ recomputation of t

$$t_j = \underbrace{(t_{j-1} - \overbrace{T[j] \cdot d^{m-1}}^{1}) \cdot d}_{2} + \overbrace{T[j+m-1]}^{3} \leftarrow O(1)$$

computation : $\theta(m) + \theta(m) + \theta(n-m)$

$\uparrow$ for P, $\uparrow$ for first T, $\uparrow$ remaining T

① $\longrightarrow$ $T[j] * d^{m-1}$ ← oldest

$d * ht - T[j] * d^m$ ← oldest_removed+adv

$+ T[j+m-1]$ ← new_added

← problem w/ Robin-Karp
if the pattern is too long, the resulting number will be huge and can cause overflow

soln: Hashing
↓
take the mod w/ a prime number q
↓
large to prevent multiple hashing

① hp = pattern mod q

② ht = (text first m) mod q

③ if (ht = hp) : compare windows, if equal return

else. if not eof. advance window

④ :C exit.

ⓐ the hash method

$(x+y) \bmod q = x \bmod q + y \bmod q$

$xy \bmod q = [(x \bmod q)(y \bmod q)] \bmod q$

$(3*10+6) \bmod 13 = ((3*10) \bmod 13 + 6 \bmod 13) \bmod 13$ mod

$= (((3 \bmod 13)*(10 \bmod 13) \bmod 13 + 6 \bmod 13)13$

$= (4 + 6 \bmod 13) \bmod 3$

$= 10$

$$(((a * 10 + d) * 10 + c) \bmod q$$

$\bmod q * (\bmod q) \bmod q$

```
int hash( Txt , m, d )
{
    int h = Txt[0] % q ;
    for ( int i = 1 ; i < m ; i++ )          ← θ(m)
        h = (h * d + Txt[i]) % q ;
    return h ;
}
```

prevents overflow

(b)

```
int rehash ( T, j, m, ht ) {
    oldest = ( T[i] * dM ) % q ;          // dM = d^(m-1) mod q
    oldest_removed = (( ht + q) - oldest ) % q ;
    return (oldest_removed * d + T[i+m] ) % q ;
}
```

$$t_j = \left(t_{j-1} - \underbrace{T[j] \cdot d^{m-1}}_{\text{oldest}\ \%q}\right) \cdot d + T[j+m-1]$$

oldest

oldest_removed %q

%q

```
int RKscan (P,T) {
    m = length(P)
    n = length(T)
    dM = 1;
    for j = 1 -> m-1    dM = dM*d % q;     } Θ(m-1)

    hp = hash(P, m, d);    Θ(m-1)
    ht = hash(T, m, d);    Θ(m-1)

n-m+1         for (j=0; j <= n-m ; j++)
iterations        if ( hp == ht && equal_string(P,T,0,j,m))
                      return j;
                  if (j < n-m)  ht = rehash(T, j, m, ht);
    }                                  Θ(1)
    return -1;
```

RK:

→ $\Theta((n-m+1)m)$: worst case

→ $O(n+m)$ + time for spurious hits ← real case

- $O(m)$ for 2 hash calls.
- $O(n)$ for loop

→ number of spurious hits can be kept low by using a large prime number q for the hash function

③ Boyer-Moore Algorithm

- text T w/ n char   T[1] is first
  pattern P w/ m char.   P[1] is first

- T from left to right.
  P from right to left.

preprocessing step : generates two tables based on which we slide the pattern as much as possible after a mismatch

efficiency comes from here

↘ improves with pattern size

```
int BMscan ( char[] P, char[] T , int m , int[] charJump,
                 int[] matchJump ) {

    int j ; int k ;
    j = m ;  k = m ;        (right to left)
    while ( j <= n ) {
        if ( k < 1 ) return j + 1 ;    match!
        if ( T[j] == P[k] ) { j-- ; k-- ; }

        else { j += max(charJump[T[j]] , matchJump[k]);
               k = m ; }
    }
    return -1 ;
}
```

starting index is 1

shift window

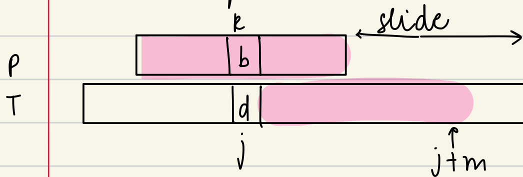
reset pattern pointer

(j + m <= n)

(j < n - m + 1)

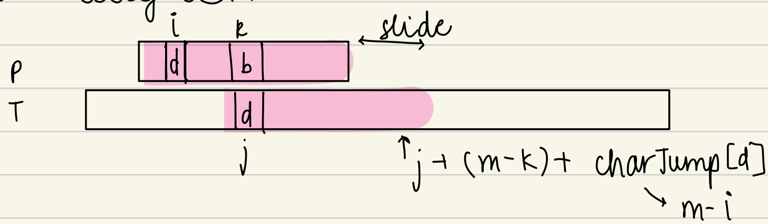
if nothing : $j = j + m$
if in array : $j = j + \text{charjump[char]}$ ← $m-k$ ← starts from 1

ⓐ CharJump



if d doesn't appear in P at all, line up P after T[j]

if d does appear in P, d in P & d in T(rightmost) are aligned.




```
void computeJumps (char [] P, int m, int alpha, int[] charJump)
{
    char ch ; int k ;                      → number of unique chars
    for ( ch = 0 , ch < alpha ; ch++)      O(alpha)
        charJump [ch] = m ;
    for ( k = 1 ; k <= m ; k++ )           O(m)
        charJump[P[k]] = m-k ;             ← position from end.
}
```

$O(\Sigma + m)$

this can fail if the index of rightmost 'd' is greater than k (moves it back)

↑ check if $\underbrace{m-k+1}_{\text{full slide}}$ is greater and then shift

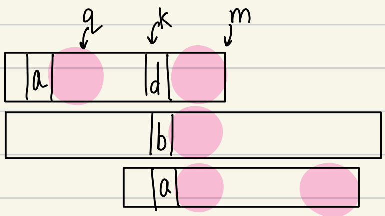
a	m-1	5
b	m-2	4
c	m-3	3
d	m-4	2
e	m-5	1
f	m-6	0

vs $\underline{\underline{m+k-1}}$

(b) matchJump

this heuristic tries to derive the maximum shift from the structure of the pattern.
defined for each char in P (diff from charJump)

case 1 :

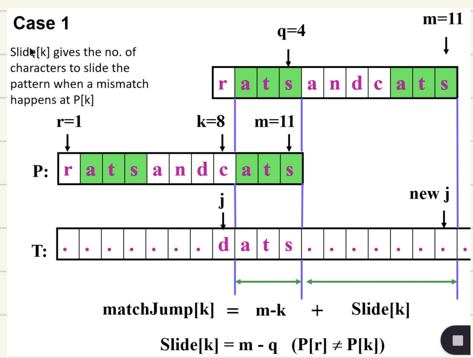


: matching suffix
① it occurs again in P
② the preceeding character is different

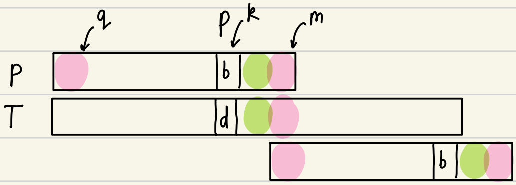
$$j = \underbrace{m-k}_{\text{slide by suffix length (j aligned w/ end of P)}} + \underline{slide[k]} \rightarrow m-q \rightarrow \text{slide by dist up to } q$$

slide by
suffix length
(j aligned w/ end of P)

(again first character is at P[1])



case 2

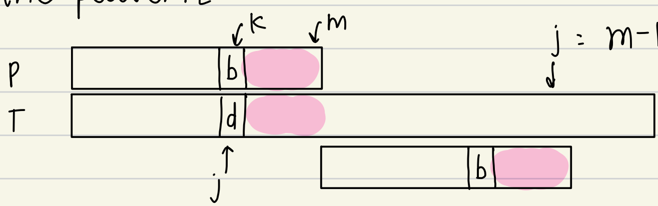


only a part of the matching suffix is at the beginning of the pattern (prefix)

$j = m-k + \text{slide}[k]$

$\hookrightarrow m-q$

Case 3 : there is no other occurence of the matching suffix in the pattern



$j = m-k + \underbrace{\text{slide}[k]}$

$\hookrightarrow m \quad (q=0)$