Pattern Matching Algorithms

- **Simple** problem: finding the position of a substring (pattern) in a string.
- **Complex** problem: matching a regular expression in a string, like m*?[rt]. It's an adaptation of the simple problem.
- Brute force approach: complexity O(nm), where n is the length of the string and m is the length of the pattern.
- Better algorithms can be of complexity O(n+m).

Brute Force Algorithm

```
int Search(char *str, char *pattern) {
  bool found=false;
  int n=strlen(str), m=strlen(pattern);
  for (int i=0; i<=n-m; i++) {
    found = true;
    for (int j=0; j<m && found; j++)
      if (str[i+j] != pattern[j])
        found = false;
    if (found)
      return i;
  }
  return -1;
}
```

The Rabin-Karp Algorithm

- Before it checks if the pattern is matched at a particular position, it does some preliminary checking if it's possible for the pattern to match or not.
- It checks for a "fingerprint" that can be easily computed and that eliminates many positions in the string.
- If the string and pattern are made of 0/1, the fingerprint can be the bit parity.
- With a good fingerprint, it can be O(n+m) on average (not in the worst case).


```
int Rabin_Karp(char *str, char *pattern) {
  bool found=false;
  int n=strlen(str), m=strlen(pattern);
  int pfinger = fingerprint(pattern, m);
  int sfinger = fingerprint(str, m);
  for (int i=0; i<=n-m && !found; i++) {
    if (pfinger == sfinger) {
      found = true;
      for (int j=0; j<m && found; j++)
        if (str[i+j] != pattern[j])
          found = false;
      if (found) return i;
    }
    sfinger = updatefinger(sfinger,str[i],str[i+m]);
  }
  return -1;
}
```

Fingerprint

- Parity:

```
fingerprint(str,m)=sum(str[i],
                    i = 0 to m-1)%2;
update(finger, old_bit, new_bit) =
  (finger+old_bit+new_bit)%2;
```

- Only eliminates on average half of the positions.
- It's better to have a fingerprint that gives a lot of different values.

Better Function

- Let q be a prime number > mn^2 and $r = 2^{m-1} \% q$.

```
int fingerprint(char *str, int m) {
  int finger = 0;
  for (int i=0; i<m; i++)
    finger = mod((finger*2 +str[i]), q);
  return finger;
}

int update(int finger, char old_bit,
           char new_bit)
{ return mod((2*(finger-r*old_bit)+new_bit),q); }

int mod(int n,int m)
{ return (n>=0 ? n%m : n%m+m); }
```

The Knuth-Morris-Pratt Algorithm

- Based on the idea that when we have a mismatch at a position, we can eliminate a good number of adjacent positions based on where could the word start from again and on the internal repetitions in the pattern.
- It starts by computing a table of number of positions to be skipped based on how many letters from the pattern have been matched in the current step.
- Complexity: O(n+m).
- Let k be such that pattern[0..k] has been matched in the string.

Examples

		t	o	r	t	o	i	s	e
k	-1	0	1	2	3	4	5	6	7
shift	1	1	2	3	3	3	6	7	8

		t	w	e	e	d	l	e	d	e	e
k	-1	0	1	2	3	4	5	6	7	8	9
shift	1	1	2	3	4	5	6	7	8	9	10

		p	a	p	p	a	r
k	-1	0	1	2	3	4	5
shift	1	1	2	2	3	3	6

		t	a	r	t	a	r
k	-1	0	1	2	3	4	5
shift	1	1	2	3	3	3	3

```
int KMP(char *str, char *pattern)
{
  int n=strlen(str), m=strlen(pattern);
  int *shift = compute_shift(pattern);
  int i=0, j=0;
  while (i<=n-m) {
    while (str[i+j] == pattern[j]) {
      j++;
      if (j >= m)
        return i;
    }
    i = i+shift[j-1];
    j = max(j-shift[j-1], 0);
  }
  return -1;
}
```



```
int *compute_shift(char *p) {
  int m=strlen(p), i=1, j=0;
  int *shift=new int[m+1];
  shift[-1] = 1; // this could be position m
  shift[0] = 1;
  while (i+j<m)
    if (p[i+j] == p[j]) {
      shift[i+j] = i;
      j = j+1;
    }
    else {
      if (j == 0)
        shift[i] = i+1;
      i = i + shift[j-1];
      j = max(j-shift[j-1], 0);
    }
}
```