Chương 4

Chiến lược biến thể-để-trị (transform-and-conquer)

Nội dung

- Chiến lược Biến thể-để-trị
- Giải thuật Gauss để giải hệ phương trình tuyến tính
- Cấu trúc heap và heapsort
- Giải thuật Horner để định trị đa thức
- So trùng dòng ký tự bằng giải thuật Rabin-Karp

1.Biến thể để trị (transform-and-conquer)

- **Kỹ thuật *biến thể-để-trị* thường làm việc theo hai bước.**
  - **Bước 1 là bước *biến thể*, thể hiện của bài toán được biến đổi để chuyển sang một dạng dễ dẫn đến lời giải.**
  - **Bước 2 là bước *tìm ra lời giải* cho bài toán.**
- **Có nhiều biến dạng của bước 1:**
  - Biến thể để đưa đến một thể hiện đơn giản hơn của bài toán (*đơn giản hóa thể hiện* -instance simplification)
  - Biến thể để đưa đến một biểu diễn khác của cùng bài toán (*biến đổi biểu diễn* -representation change)
  - Biến thể để đưa đến một thể hiện của một bài toán khác mà đã có tồn tại giải thuật (*thu giảm bài toán* - problem reduction).

2. Giải thuật Gauss để giải hệ phương trình tuyến tính

- Cho hệ phương trình gồm *n* phương trình với *n* ẩn số.

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$$

- Để giải hệ phương trình trên, ta dùng giải thuật **loại trừ Gauss** (Gauss elimination).
- Ý tưởng chính của giải thuật : biến đổi hệ thống *n* phương trình tuyến tính với *n* biến thành một hệ thống tương đương (tức là có cùng lời giải như hệ phương trình ban đầu) với một ma trận *tam giác trên* (một ma trận với các hệ số zero dưới đường chéo chính)
- Giải thuật Gauss thể hiện tinh thần của chiến lược biến thể-để-trị theo kiểu "đơn giản hóa thể hiện" (instance simplification).

$$
\begin{array}{ll}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 & a'_{11}x_1 + a'_{12}x_2 + \ldots + a'_{1n}x_n = b'_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 & a'_{22}x_2 + \ldots + a'_{2n}x_n = b'_2 \\
\quad ; & \Rightarrow \\
\quad ; & \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n & a'_{nn}x_n = b'_n
\end{array}
$$

Làm cách nào để ta có thể chuyển một hệ thống với ma trận A bất kỳ thành một hệ thống tương đương với ma trận tam giác trên A'?

Bằng một loạt các phép biến đổi cơ bản như sau:

- Hoán vị hai phương trình trong hệ thống
- Thay một phương trình bằng phương trình đó nhân với một hệ số.
- Thay một phương trình với tổng hay hiệu phương trình đó với một phương trình khác được nhân một hệ số.

Thí dụ

$2x_1 - x_2 + x_3 = 1$
$4x_1 + x_2 - x_3 = 5$
$x_1 + x_2 + x_3 = 0$

2	-1	1	1	
4	1	-2	5	row 2 – (4/2) row 1
1	1	1	0	row 3 – (1/2) row 1

2	-1	1	1	
0	3	-3	3	row 3 – (1/2) row 2
0	3/2	½	-1/2	

2	-1	1	1	
0	3	-3	3	
0	0	2	-2	$\Rightarrow$ $x_3 = (-2/2)=1$; $x_2 = (3-(-3)x_3/3 = 0$;
				$x_1 = (1-x_3 - (-1)x_2)/2 = 1$

Giải thuật Gauss

```
GaussElimination(A[1..n,1..n],b[1..n])
for i := 1 to n do A[i,n+1] := b[i];
for i := 1 to n -1 do
  for j := i + 1 to n do
    for k:= i to n+1 do
      A[j,k] := A[j,k]-A[i,k]*A[j,i]/A[i,i];
```

Lưu ý: Vector cột *b* cũng được gom vào thành cột thứ n+1 của ma trận *A*.

Trở ngại: Khi A[i,i] = 0, giải thuật không làm việc được. Và khi |A[i,i]| quá nhỏ, giải thuật sẽ bị *sai số làm tròn* khi máy tính tính toán (round-off-error) gây ảnh hưởng xấu, làm cho sự tính toán trở nên không chính xác.

Giải thuật Gauss cải tiến

- Để tránh trường hợp |A[i,i]| quá nhỏ nêu trên, ta áp dụng kỹ thuật *tìm phần tử chốt bán phần* (partial pivoting) được mô tả như sau:

  "*Tại lượt lặp thứ i của vòng lặp ngoài, ta cần tìm hàng nào có hệ số ở cột thứ i mang giá trị tuyệt đối lớn nhất và hoán đổi hàng này với hàng i và dùng hệ số đó như là phần tử chốt của lượt lặp thứ i*"

Giải thuật Gauss cải tiến

```
BetterGaussElimination(A[1..n,1..n],b[1..n])
for i := 1 to n do A[i,n+1] := b[i];
for i := 1 to n -1 do
   pivotrow := i;
   for j := i+1 to n do
      if |A[j,i]| > |A[pivotrow,i]| then pivotrow:= j;
   for k:= i to n+1 do
      swap(A[i,k], A[pivotrow,k]);
   for j := i + 1 to n do
      temp:= A[j,i]/A[i,i];
      for k:= i to n+1 do
        A[j,k] := A[j,k]-A[i,k]*temp;
```

Độ phức tạp của giải thuật Gauss

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+1-i+1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n+2-i)$$

$$= \sum_{i=1}^{n-1} (n+2-i)(n-(i+1)+1) = \sum_{i=1}^{n-1} (n+2-i)(n-i)$$

$$= (n+1)(n-1)+n(n-2)+..+3.1$$

$$= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j$$

$$= (n-1)n(2n-1)/6 + 2(n-1)n/2$$

$$= n(n-1)(2n+5)/6 \approx n^3/3 = O(n^3)$$

Sau khi dùng giải thuật Gauss để đưa ma trận về dạng ma trận tam giác trên, ta sẽ dùng phương pháp thay thế lùi (backward subsitution) để tính ra giá trị của các ẩn.

3. Cấu trúc dữ liệu heap và heapsort

Hàng đợi có độ ưu tiên (a *priority-queue*) là cấu trúc dữ liệu mà hỗ trợ ít nhất hai tác vụ:

- thêm một phần tử mới vào cấu trúc
- Tìm phần tử có độ ưu tiên lớn nhất
- xóa bỏ phần tử có độ ưu tiên lớn nhất

Hàng đợi có độ ưu tiên khác với hàng đợi thông thường ở điểm khi lấy phần tử ra khỏi hàng đợi thì đó không phải là phần tử *cũ nhất* trong hàng đợi mà là phần tử *có độ ưu tiên lớn nhất* trong hàng đợi.

Thi công hàng đợi có độ ưu tiên

Hàng đợi có độ ưu tiên như đã mô tả là một ví dụ về kiểu dữ liệu trừu tượng. Có hai cách để thi công hàng đợi có độ ưu tiên:

1. Dùng mảng để thi công hàng đợi có độ ưu tiên (Cách này thì đơn giản khi thêm vào một phần tử mới nhưng khi xóa bỏ phần tử có độ ưu tiên lớn nhất ra khỏi hàng đợi thì độ phức tạp sẽ cao.)
2. Dùng cấu trúc dữ liệu heap.

Cấu trúc dữ liệu heap

Cấu trúc dữ liệu mà có thể hỗ trợ cho các tác vụ làm việc với *hàng đợi có độ ưu tiên* sẽ chứa các mẩu tin trong một mảng sao cho:

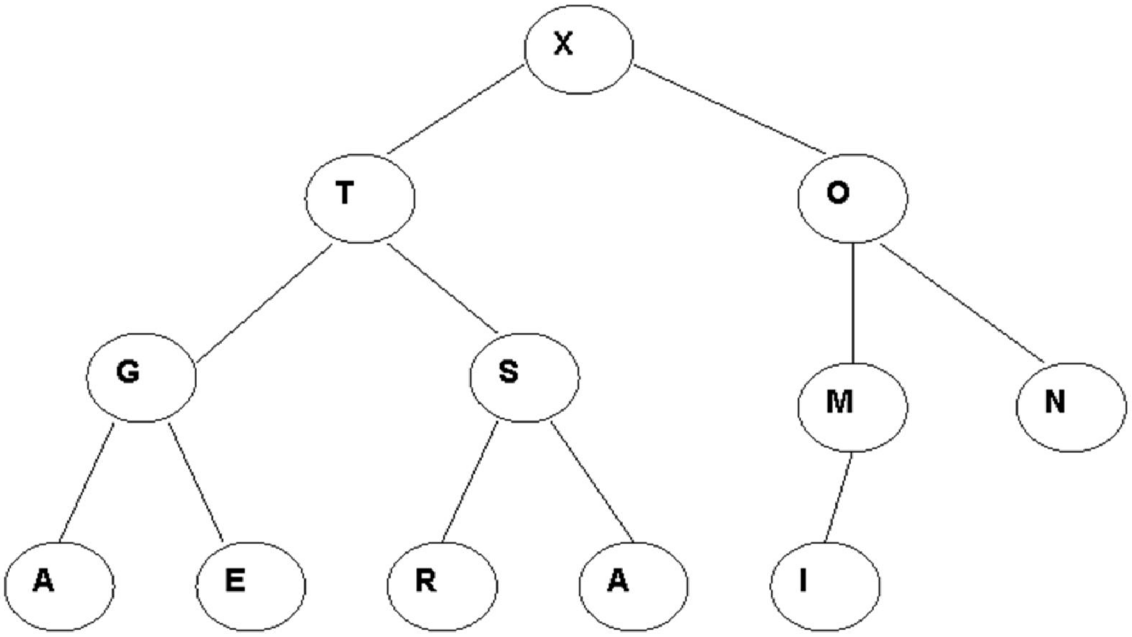
mỗi khóa phải lớn hơn khóa ở hai vị trí khác trong mảng. Tương tự mỗi khóa trong hai khóa này phải lớn hơn hai trị khóa khác và cứ như thế..

Thứ tự này sẽ dễ thấy hơn khi ta diễn tả mảng như một cấu trúc cây với những đường nối mỗi khóa xuống hai khóa nhỏ hơn.

Các trị khóa trong cấu trúc cây thỏa điều kiện *heap* như sau:

Khóa tại mỗi nút cần phải lớn hơn (hay bằng) các khóa ở hai con của nó (nếu có). Điều này hàm ý trị khóa lớn nhất ở nút rễ.

Thí dụ: Heap dưới dạng cây nhị phân



k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	X	T	O	G	S	M	N	A	E	R	A	I

Heap dưới dạng một mảng

- Ta có thể diễn tả dạng cây của heap thành một mảng bằng cách đặt nút rễ tại vị trí 1 của mảng, các con của nó tại vị trí 2 và 3, các nút ở các mức kế tiếp ở các vị trí 4, 5, 6 và 7, v.v..

k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	X	T	O	G	S	M	N	A	E	R	A	I

Từ một nút dễ dàng để đi tới nút cha và các nút con của nó.

- **Cha một nút ở vị trí *j* sẽ là nút ở vị trí *j div 2*.**
- **Hai con của một nút ở vị trí *j* sẽ ở các vị trí *2j* và *2j+1*.**

Các lối đi trên heap

- Một heap là một cây nhị phân, được diễn tả như là một mảng trong đó mỗi nút thỏa mãn điều kiện heap. Đặc biệt, phần tử có khóa lớn nhất luôn ở *vị trí thứ nhất* của mảng.

- Tất cả các giải thuật làm việc trên heap đi dọc theo một lối đi nào đó từ nút rễ xuống mức đáy (bottom) của heap.

  $\Rightarrow$ Trong một heap có N nút, tất cả các lối đi (path) thường có lgN nút trên đó.

Các giải thuật trên Heap

Có hai tác vụ quan trọng làm việc trên heap: thêm vào phần tử mới và xóa bỏ phần tử lớn nhất ra khỏi heap.

1. Tác vụ thêm vào (*insert)*

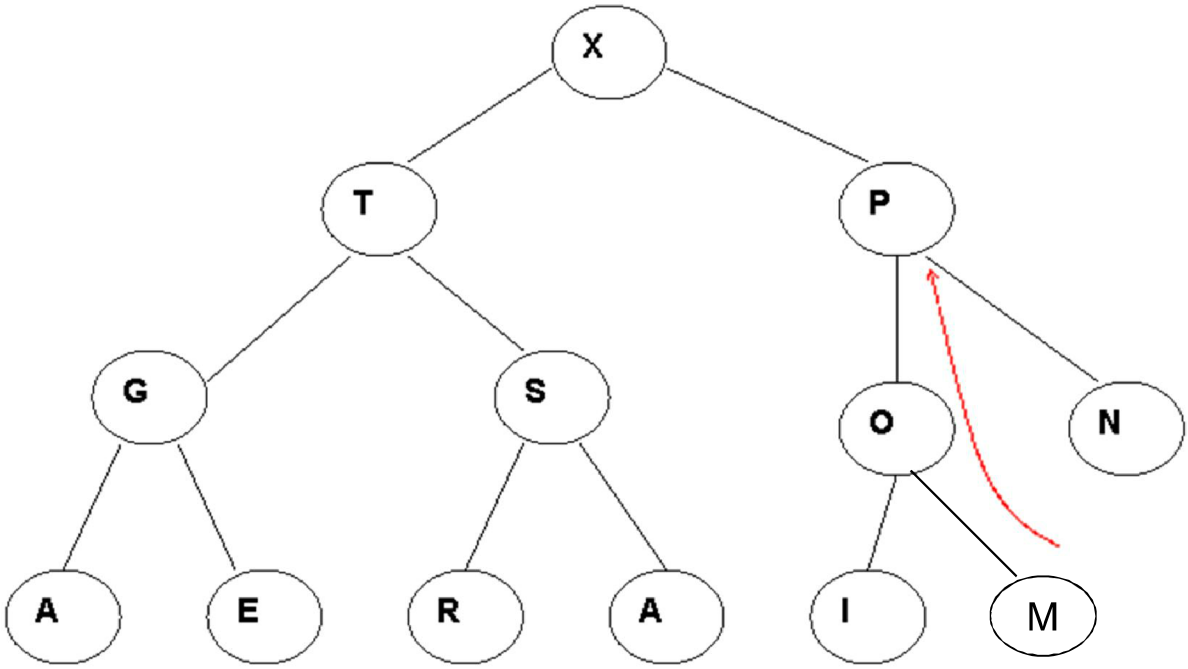
Tác vụ này sẽ làm tăng kích thước của heap lên thêm một phần tử. N được tăng thêm 1.
Và phần tử mới được đặt vào tại vị trí a[N], nhưng lúc đó điều kiện heap có thể sẽ bị vi phạm.

Nếu điều kiện heap bị vi phạm, nó sẽ được khắc phục bằng cách *hoán đổi* phần tử mới với cha của nó. Điều này lại có thể gây ra vi phạm điều kiện heap và nó sẽ được khắc phục tiếp với cùng một cách tương tự.

Tác vụ thêm vào

```
procedure upheap(k:integer)
var  v: integer;
begin
        v :=a[k]; a[0]:= maxint;
        while a[k div 2] <= v do
        begin a[k]:= a[k div 2 ]; k:=k div 2 end;
    a[k]:= v
end;
procedure insert(v:integer);
begin
        N:= N+1; a[N] := v ; upheap(N)
end;
```

Thêm (P) vào heap



Tác vụ xóa bỏ phần tử lớn nhất

- Tác vụ xóa sẽ làm giảm kích thước của heap một đơn vị, tức nó làm giảm N một đơn vị.

- Nhưng phần tử lớn nhất (tức *a[1]*) sẽ được xóa bỏ và được thay thể bằng phần tử mà đã ở vị trí *a[N]*. Nếu trị khóa tại nút rễ quá nhỏ, nó phải được *di chuyển xuống* để thỏa mãn điều kiện heap.

- Thủ tục *downheap* thực hiện việc di chuyển phần tử đang ở nút rễ xuống bằng cách hoán đổi nút ở vị trí *k* với nút lớn hơn trong hai nút con của nó, nếu cần và dừng lại khi nút ở *k* lớn hơn hai nút con của nó.

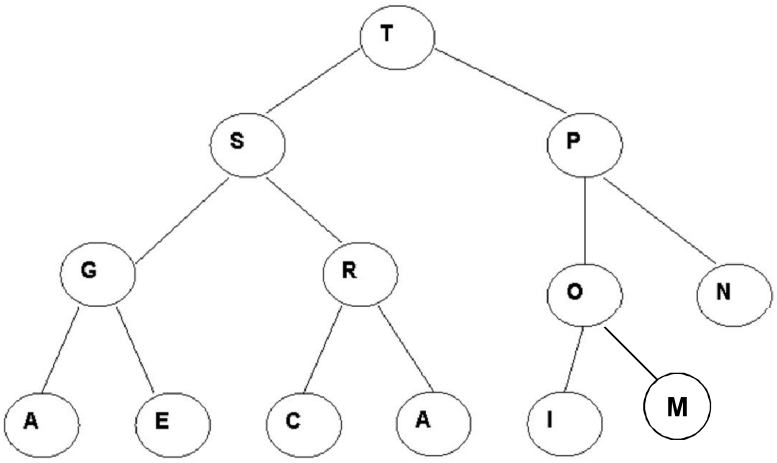
Tác vụ xóa bỏ

```
procedure downheap(k: integer);
label  0 ;
var  j, v : integer;
begin
    v:= a[k];
    while k<= N div 2 do
    begin
        j:= 2*k;
        if j < N then if a[j] < a[j+1] then
                        j:=j+1;
        if v >= a[j] then go to 0;
        a[k]:= a[j]; k:= j;
    end;
0: a[k]: =v
end;
```

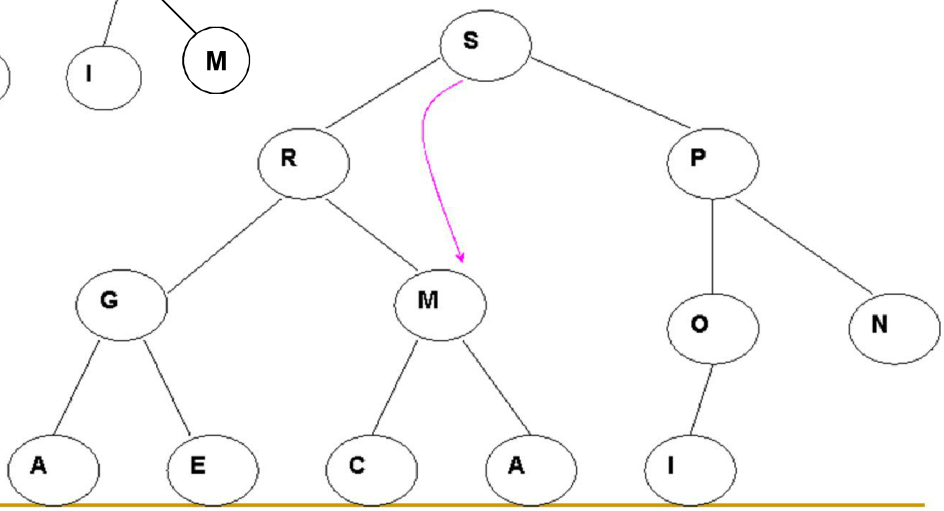
```
function remove: integer;
begin
    remove := a[1];
    a[1] := a[N]; N := N-1;
    downheap(1);
end;
```

Thí dụ về tác vụ xóa

Trước khi xóa



Sau khi xóa



Độ phức tạp của các tác vụ trên heap

<u>Tính chất 3.1:</u> ***Mọi tác vụ thêm vào, xóa bỏ, downheap, upheap đòi hỏi ít hơn 2lgN so sánh khi thực hiện trên một heap gồm N phần tử.***

Tất cả những tác vụ này phải đi dọc theo một lối đi giữa nút rễ cho đến cuối heap mà bao gồm ít hơn lgN phần tử với một heap gồm N phần tử.

Thừa số 2 là do tác vụ downheap khi xóa bỏ mà cần hai thao tác so sánh trong vòng lặp trong và các thao tác khác chỉ đòi hỏi lgN lần so sánh.

Giải thuật heapsort

Ý tưởng: Giải thuật bao gồm 2 công tác (1) tạo một heap chứa những phần tử cần sắp thứ tự và (2) lần lượt lấy chúng ra khỏi heap theo một thứ tự.

M : kích thước của heap

N: số phần tử cần được sắp thứ tự.

```
N:=0;
for k:= 1 to M do
   insert(a[k]); /* construct the heap */
for k:= M downto 1 do
    a[k]:= remove;    /*putting the element removed into the
                        array a */
```

Độ phức tạp của heap sort

Tính chất: *Heapsort dùng ít hơn 3MlgM lần so sánh để sắp thứ tự M phần tử.*

Giới hạn trên này xuất phát từ giải thuật heapsort và tính chất của hai tác vụ thêm vào/xóa bỏ trên heap.

Vòng *for* thứ nhất tốn MlgM lần so sánh.

Vòng *for* thứ hai tốn 2MlgM lần so sánh.

Tổng cộng:

MlgM + 2MlgM = 3MlgM.

Heapsort là một thí dụ điển hình của chiến lược *Biến thể-để-trị*, dùng kỹ thuật "biến đổi biểu diễn" (representation change)

4. Giải thuật Horner để định trị đa thức

Ta cần định trị đa thức sau

$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ (4.1)

tại một điểm x.

G.H Horner, một nhà toán học người Anh, cách đây 150 năm đã đưa ra một qui tắc đinh trị đa thức rất hữu hiệu.

Qui tắc Horner là một thí dụ tốt về chiến lược *Biến thể-để-trị* dùng kỹ thuật "thay đổi biểu diễn" (representation change).

Từ công thức 4.1 ta có thể thu được một công thức mới bằng cách liên tiếp dùng *x* làm thừa số chung trong những đa thức con còn lại với số mũ giảm dần.

$p(x) = (\ldots(a_n x + a_{n-1})x + \ldots)x + a_0$ (4.2)

Giải thuật Horner

```
Horner(P[0..n],x)
// Mảng P[0..n] chứa các hệ số của đa thức.
p := P[n];
for j := n -1 down to 0 do
    p := p*x + P[i];
return p;
```

Tổng số phép nhân và tổng số phép cọng trong giải thuật chỉ là *n*.

Trong khi đó nếu tính trực tiếp đa thức thì chỉ riêng số hạng $a_n x^n$ đã cần đến *n* phép nhân.

⇒ **Giải thuật Horner là giải thuật tối ưu để định trị đa thức.**

5. So trùng dòng ký tự bằng giải thuật Rabin-Karp

- ***So trùng dòng ký tự***: tìm tất cả sự xuất hiện của một khuôn mẫu (pattern) trong một văn bản (text).
- Văn bản là một mảng ký tự T[1..n] chiều dài *n* và một khuôn mẫu là một mảng P[1..m] chiều dài *m*.
- Các phần tử của P và T là những ký tự lấy từ một *tập ký tự* (*alphabet)* Σ.
- Khuôn mẫu P xuất hiện với bước dịch chuyển(shift) *s* trong văn bản T (tức là, P xuất hiện bắt đầu từ vị trí *s+1* trong văn bản T) nếu $1 \le s \le n - m$ và T[s+1..s+m] = P[1..m].
- Nếu P xuất hiện với bước dịch chuyển *s* trong T, thì ta bảo *s* là một *bước dịch chuyển hợp lệ (valid shift)*; ngược lại ta bảo *s* là một *bước dịch chuyển không hợp lệ* (*invalid shift)*.

- **Bài toán so trùng dòng ký tự là bài toán tìm tất cả những bước dịch chuyển hợp lệ mà một khuôn mẫu *P* xuất hiện trong một văn bản *T* cho trước.**

Văn bản **abcabaabcabac**

Khuôn mẫu **abaabcabac**

Bước dịch chuyển: $s = 3$

Giải thuật Rabin-Karp vận dụng những khái niện căn bản trong lý thuyết số chẳng hạn sự tương đương của hai số modulo một số thứ ba.

Giải thuật Rabin-Karp

- Giả sử Σ = {0, 1, 2, …, 9}, tức mỗi ký tự là một ký số thập phân. (Trong trường hợp tổng quát, mỗi ký tự là một ký số của cơ hệ *d*, tức là *d* = $|\Sigma|$.)
- Ta có thể xem một dòng gồm *k* ký tự kế tiếp diễn tả một số thập phân có chiều dài *k*. Dòng ký tự "31415" tương ứng với trị số thập phân 31415.
- Cho một khuôn mẫu P[1..m], gọi *p* là giá trị thập phân tương ứng với khuôn mẫu.
- Cho một văn bản T[1..n], gọi t_s là trị số thập phân của dòng con chiều dài *m* T[s+1...s+m], với *s* = 0, 1, …, *n-m*.
- t_s = *p* nếu và chỉ nếu T[s+1..s+m] = P[1..m] và *s* là một bước dịch chuyển hợp lệ nếu và chỉ nếu t_s = *p*

- Ta có thể tính *p* trong thời gian O(m) dùng qui tắc Horner:

  $p = P[m] + 10*(P[m-1] + 10*(P[m-2] + \ldots$
  $+ 10*(P[2] + 10*P[1])\ldots))$

- Giá trị t_0 có thể được tính một cách tương tự từ T[1..m] trong thời gian O(m).
- Chú ý: t_{s+1} có thể được tính từ t_s:

  $t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$ (5.1)

<u>Thí dụ</u>: Nếu m = 5 và t_s = 31415, thì ta sẽ bỏ ký số bậc cao T[s+1] = '3' và đưa vào ký số bậc thấp là '2' để đạt giá trị:

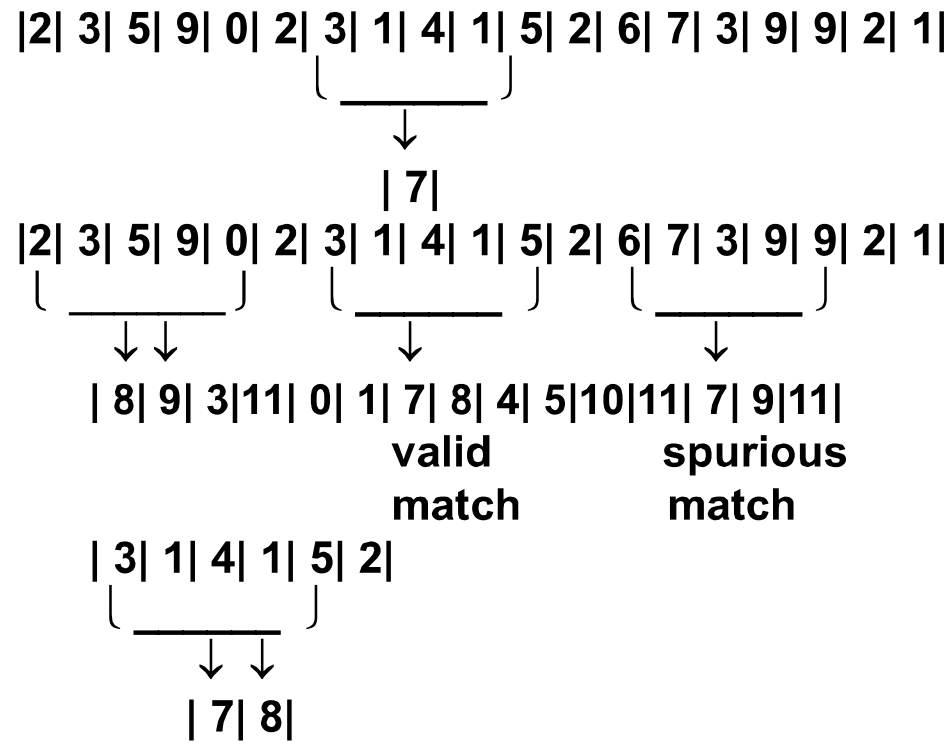
$t_{s+1} = 10(31415 - 10000.3) + 2 = 14152$

- Mỗi lần thực thi phương trình (5.1) sẽ cần tiến hành một số lượng phép toán số học cố định.
- Việc tính toán t_1, t_2,…, t_{n-m} tỉ lệ với O(n-m).
- Như vậy, p và t_0, t_1, …, t_{n-m} có thể được tính trong chi phí thời gian O(m) +O(m) + O(n-m) ≈ O(n + m).
- Nhưng *p* và t_s có thể quá lớn đến nỗi máy tính không thể biểu diễn được. Để khắc phục vấn đề này, ta tính *p* và các t_s modulo một đại lượng *q* thích hợp.
- Đại lượng *q* thường được chọn là một số nguyên tố sao cho *10q* thì chứa được trong một từ của máy tính.
- Trong trường hợp tổng quát, với bộ mẫu tự gồm *d* ký tự {0, 1, …, d-1}, ta chọn *q* sao cho *dq* chứa được trong một từ của máy tính.

- Và phương trình (5.1) trở thành:
  $t_{s+1} = d(t_s - hT[s+1]) + T[s+m+1])\bmod q$ (5.2)
  với $h = d^{m-1} \pmod q$
- Tuy nhiên, $t_s \equiv p \pmod q$ không hàm ý $t_s = p$.
- Mặt khác, nếu $t_s \neq p \pmod q$ thì ta có thể khẳng định $t_s \neq p$, và như vậy bước dịch chuyển *s* là không hợp lệ.
- Chúng ta có thể dùng cách thử $t_s \equiv p \pmod q$ để loại bỏ những *bước dịch chuyển không hợp lệ s*.
- Một bước dịch chuyển *s* mà thỏa $t_s \equiv p \pmod q$ thì phải được thử nghiệm thêm để xem s có thực sự là bước dịch chuyển hợp lệ hay chỉ là một sự khớp trùng giả (*spurious hit)* mà thôi .

Giải thuật Rabin-Karp thể hiện rõ nét tinh thần chiến lược Biến thể-để-trị

Thí dụ:



$14152 = (31415 - 3 \times 1000) \times 10 + 2 \pmod{13}$

$= 8 \pmod{13}$

```
procedure RABIN-KARP-MATCHER(T, P, d, q);
/* T is the text, P is the pattern, d is the radix and q is the prime */
begin
    n: = |T|; m: = |P|;
    h: = d^(m-1) mod q;
    p: = 0; t0: = 0;
    for i: = 1 to m do
    begin
        p: = (d*p + P[i])mod q;
        t0: = (d*t0 + T[i])mod q
    end
    for s: = 0 to n – m do
    begin
        if p = ts then  /* there may be a hit  */
        if P[1..m] = T[s+1..s+m] then
            Print "Pattern occurs with shift "s;
        if s < n – m then
            ts+1: = (d(ts –T[s + 1]h) + T[s+m+1])mod q
    end
end
```

Thời gian thực thi của RABIN-KARP-MATCHER là O((n – m + 1)m) trong trường hợp xấu nhất vì khi đó giải thuật phải kiểm tra lại mọi bước dịch chuyển hợp lệ.

Trong nhiều ứng dụng, thường chỉ có một vài bước dịch chuyển hợp lệ và do đó thời gian chạy thường là O(n+m) cọng với thời gian đòi hỏi để kiểm tra lại các sự khớp trùng giả.

Vài ghi nhận về chiến lược biến thể-để-trị

- **Cây AVL** là cây tìm kiếm nhị phân mà luôn luôn được làm cho cân bằng.
  - Sự cân bằng này được duy trì bằng 4 *phép quay* (rotation).
  - Tất cả các thao tác trên cây AVL đều có độ phức tạp O(nlgn), loại trừ được trường hợp xấu nhất của cây tìm kiếm nhị phân.
- **Cây AVL** và **giải thuật loại trừ Gauss** là những thí dụ của **biến thể-để-trị** theo kiểu "*đơn giản hóa thể hiện*".
- **Heapsort**, **giải thuật Horner** và **giải thuật Rabin-Karp** là những thí dụ của **biến thể-để-trị** theo kiểu "*biến đổi cách biểu diễn*"