

Chương 1. CÂY HẬU TỔ VÀ MỘT SỐ ỨNG DỤNG TRONG XỬ LÝ XÂU

Lê Minh Hoàng (ĐHSPHN)

Cây hậu tố là một cấu trúc dữ liệu biểu diễn các hậu tố của một chuỗi, được ứng dụng rộng rãi trong các thuật toán xử lý chuỗi bởi nó cung nhiều phép toán hiệu quả giúp làm giảm thời gian thực hiện giải thuật. Một cấu trúc dữ liệu dẫn xuất là mảng hậu tố tuy phạm vi ứng dụng hẹp hơn cây hậu tố nhưng lại rất đơn giản trong cài đặt. Tận dụng những ưu điểm của cả hai cấu trúc dữ liệu đó, rất nhiều thuật toán hiệu quả đã được công bố trong những năm gần đây. Chuyên đề này giới thiệu một số thuật toán xây dựng cây hậu tố và mảng hậu tố cùng với một số bài toán cơ bản mà thuật toán giải quyết chúng là những ví dụ điển hình của việc ứng dụng hai cấu trúc dữ liệu này. Bên cạnh đó, chuyên đề cũng trình bày một số mở rộng và thảo luận về kinh nghiệm cài đặt trong các kỳ thi lập trình với thời gian hạn chế.

1.1. Giới thiệu

Cây hậu tố (*suffix trees*) là một cấu trúc dữ liệu quan trọng được sử dụng trong rất nhiều thuật toán xử lý chuỗi. Sức mạnh của cây hậu tố nằm ở khả năng biểu diễn tất cả các hậu tố của một chuỗi và cung cấp nhiều phép toán quan trọng giúp nâng cao tính hiệu quả của những thuật toán. Chính nhờ những tính chất đó mà cây hậu tố được sử dụng trong rất nhiều lĩnh vực khác nhau như: xử lý văn bản, trích chọn và tìm kiếm thông tin, phân tích dữ liệu sinh học, đối sánh mẫu v.v...

Bên cạnh ưu điểm là một cấu trúc dữ liệu mạnh, các thuật toán trực tiếp xây dựng cây hậu tố có nhược điểm là phức tạp và tốn bộ nhớ. Mảng hậu tố (*suffix arrays*) là một cấu trúc dữ liệu dẫn xuất từ cây hậu tố và là một sự thay thế hợp lý cho cây hậu tố trong một số ứng dụng đặc thù. Xét về tính năng, mảng hậu tố không hỗ trợ nhiều phép toán như cây hậu tố nhưng lại có thể cài đặt khá dễ dàng.

Mặc dù mảng hậu tố là cấu trúc dữ liệu dẫn xuất và có thể xây dựng từ cây hậu tố tương ứng, đã có rất nhiều thuật toán có thể xây dựng mảng hậu tố một cách trực tiếp mà không cần dùng đến cây hậu tố. Những thuật toán như vậy cho phép đơn giản hóa rất nhiều thao tác xử lý chuỗi, bởi trong trường hợp có thể sử dụng mảng hậu tố để giải quyết, ta không cần biết về khái niệm cây hậu tố nữa.

Cũng từ khi có những thuật toán trực tiếp và hiệu quả xây dựng mảng hậu tố, rất nhiều nghiên cứu đã tìm ra phương pháp xây dựng theo chiều ngược lại: Dựng cây hậu tố từ mảng hậu tố. Những phương pháp này có ưu điểm là nhanh và tiết kiệm bộ nhớ hơn so với phép xây dựng trực tiếp cây hậu tố. Ngoài ra, những phương pháp này còn cung cấp nhiều kỹ thuật hay trong xử lý dữ liệu, có thể kế thừa để ứng dụng trong những lĩnh vực khác.

Trong các phần tiếp theo của chuyên đề, phần 1.2 giới thiệu các khái niệm cơ sở về trie hậu tố, cây hậu tố, mảng hậu tố và mảng tiền tố chung dài nhất. Phần 2 trình bày một số thuật toán xây dựng mảng hậu tố và cây hậu tố. Phần 3 nêu một số bài toán cơ bản cho

thấy hiệu quả của việc ứng dụng cấu trúc dữ liệu mảng hậu tố và cây hậu tố. Cuối cùng là kết luận và một số mở rộng của cấu trúc dữ liệu.

1.2. Một số khái niệm cơ sở

Gọi Σ là một tập hữu hạn có thứ tự gọi là bảng chữ cái (*alphabet*), các phần tử $\in \Sigma$ được gọi là ký tự. Σ^* là tập các xâu (*string*) gồm các ký tự $\in \Sigma$. Có thể coi mỗi xâu $\in \Sigma^*$ là một dãy hữu hạn các ký tự $\in \Sigma$. Ký hiệu ϵ là xâu rỗng, tập các xâu khác rỗng được gọi là $\Sigma^+ = \Sigma - \{\epsilon\}$.

Chiều dài của một xâu x , ký hiệu $|x|$, là số ký tự trong xâu x . Các ký tự trong xâu x được đánh số từ 0 tới $|x| - 1$: $x = x_0x_1 \dots x_{|x|-1} = x[0 \dots |x|]$.

Xâu nối của hai xâu x và y , ký hiệu xy , có chiều dài $|x| + |y|$ và tạo thành bằng cách lấy các ký tự trong x sau đó nối tiếp với các ký tự trong y .

Ta gọi xâu w là tiền tố (*prefix*) của xâu x , ký hiệu $w \sqsubset x$, nếu tồn tại xâu y để $x = wy$, xâu w được gọi là hậu tố (*suffix*) của xâu x , ký hiệu $w \sqsupset x$, nếu tồn tại xâu y để $x = yw$. Dễ thấy rằng nếu w là tiền tố hoặc hậu tố của x thì $|w| \leq |x|$. Một xâu có thể vừa là tiền tố vừa là hậu tố của một xâu khác. Ví dụ **ABA** vừa là tiền tố vừa là hậu tố của xâu **ABABA**. Xâu rỗng ϵ vừa là tiền tố, vừa là hậu tố của tất cả các xâu.

Hai quan hệ \sqsubset, \sqsupset có tính bắc cầu, tức là:

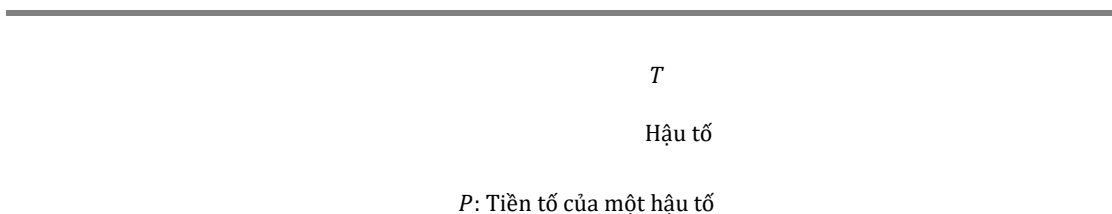
- ✳ Nếu $x \sqsubset y$ và $y \sqsubset z$ thì $x \sqsubset z$.
- ✳ Nếu $x \sqsupset y$ và $y \sqsupset z$ thì $x \sqsupset z$.

Bổ đề 1-1 (về tính gối nhau của các tiền tố và các hậu tố)

- ✳ Nếu x và y cùng là tiền tố của một xâu z thì $x \sqsubset y \Leftrightarrow |x| \leq |y|$
- ✳ Nếu x và y cùng là hậu tố của một xâu z thì $x \sqsupset y \Leftrightarrow |x| \leq |y|$

Việc chứng minh Bổ đề 1-1 khá hiển nhiên: Hai tiền tố của cùng một xâu có quan hệ tiền tố và hai hậu tố của cùng một xâu có quan hệ hậu tố.

Xâu con của một xâu T là một dãy các ký tự liên tiếp trong T . Cụ thể là nếu $T = t[0 \dots n]$ và $P = p[0 \dots m]$ là hai xâu ký tự, ta nói xâu P xuất hiện trong xâu T tại vị trí k nếu $P = t[k \dots k + m]$. Nếu xâu P xuất hiện trong xâu T ở một vị trí nào đó thì P là xâu con (*substring*) của T . Một cách định nghĩa khác về xâu con của T đó là một **tiền tố của một hậu tố** của T .



Hình 1-1. Xâu con = tiền tố của một hậu tố

Trong các ví dụ của chuyên đề này, ta coi tập chữ cái Σ là tập 26 ký tự hoa tiếng Anh: từ 'A' đến 'Z' cộng thêm một ký tự cảm canh ký hiệu '@'. Thứ tự của các ký tự giống như trong

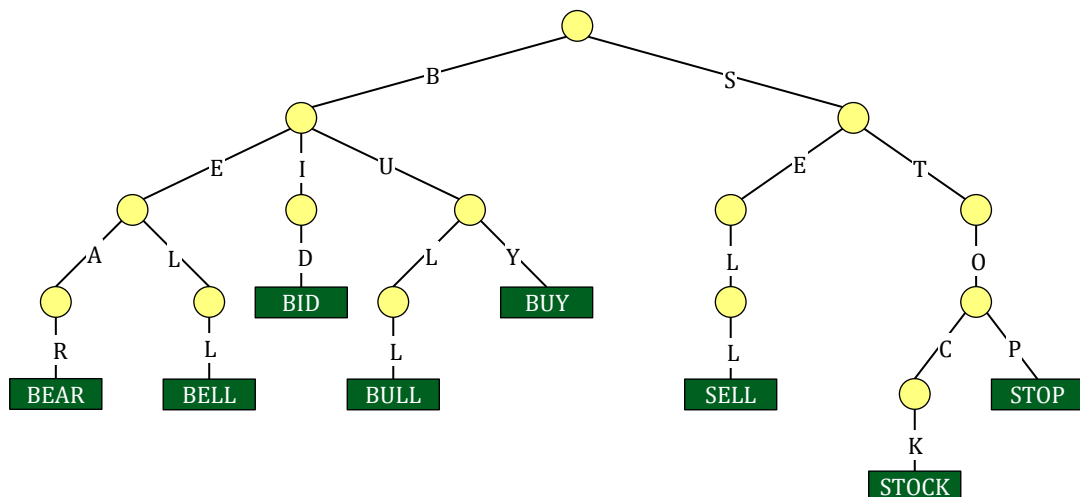
các bảng mã thông dụng ANSI/ASCII/Unicode: Ký tự '@' là ký tự đầu tiên trong bảng chữ cái Σ , tiếp theo là các ký tự từ 'A' đến 'Z'.

1.1. Trie hậu tố

Cho S là một tập gồm n xâu khác rỗng thỏa mãn: không xâu nào là tiền tố của một xâu khác. $\text{Trie}^*[1]$ của tập S là một cấu trúc dữ liệu dạng cây biểu diễn các xâu $\in S$:

- ✿ Mỗi cạnh của cây có nhãn là một ký tự $\in \Sigma$. Các cạnh đi từ một nút xuống các nút con của nó phải mang các nhãn hoàn toàn phân biệt.
- ✿ Mỗi nút v trên trie cũng mang một nhãn, nhãn của nút v , ký hiệu \bar{v} là xâu tạo thành bằng cách nối tiếp các ký tự nhãn cạnh trên đường đi từ gốc xuống nút v . Chiều dài của xâu \bar{v} được gọi là **độ sâu** của nút v , ký hiệu $\text{depth}(v)$. Theo cấu trúc của trie, hai nút khác nhau phải có xâu nhãn khác nhau.
- ✿ Có tương ứng 1-1 giữa các xâu $\in S$ với các nút lá trên trie: trie có đúng n lá và mỗi lá có nhãn là một xâu $\in S$.

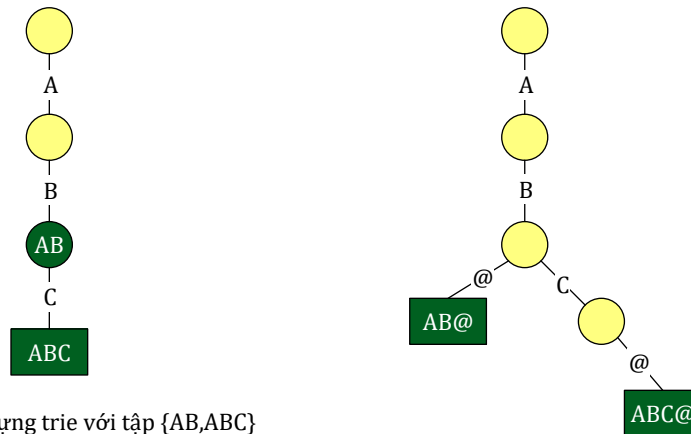
Hình 1-8 là trie biểu diễn 8 xâu: BEAR, BELL, BID, BULL, BUY, SELL, STOCK, STOP



Hình 1-2. Trie

Các xâu trong tập S phải thỏa mãn tính chất phi tiền tố (*prefix-free*): không có xâu nào là tiền tố của một xâu khác. Tính chất phi tiền tố là một điều kiện quyết định để xây dựng trie, hầu hết những thuật toán dựa trên trie đều phải ràng buộc tính chất này của dữ liệu (điển hình là thuật toán mã hóa Huffman). Một trong những kỹ thuật để đảm bảo dữ liệu có tính phi tiền tố là bổ sung một ký tự đứng đầu bảng chữ cái Σ làm **ký tự cầm canh** mà ta ký hiệu là @: mỗi xâu $\in S$ sẽ được nối thêm ký tự @ vào cuối xâu để đảm bảo không có xâu nào là tiền tố của một xâu khác. Có thể thấy rằng khi sử dụng ký tự cầm canh thì mọi cạnh trên trie nối tới nút lá đều mang nhãn @.

* Trie là thuật ngữ lấy từ retrieval, được phát âm là /'tri:/ giống như "tree" hoặc /'traɪ/ giống như "try"



Không thể dựng trie với tập {AB,ABC}
 Khi từ AB đã ứng với nút nhánh

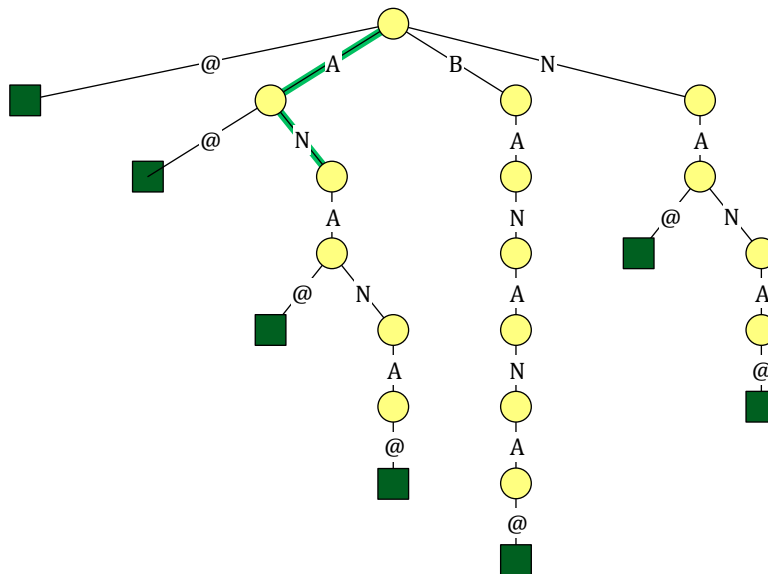
Trie với tập {AB@,ABC@}

Hình 1-3. Vai trò của ký tự cầm canh @

Nếu S là tập các hậu tố khác rỗng của một chuỗi $T \in \Sigma^+$ thì trie biểu diễn S được gọi là trie hậu tố (*suffix trie*) của T . Để thỏa mãn tính chất phi tiền tố của tập S , ta coi chuỗi T có một ký tự cầm canh @ đứng cuối cùng còn mọi ký tự khác trong T đều không phải ký tự @. Ví dụ nếu T là chuỗi BANANA@, tập S các hậu tố khác rỗng của T gồm có 7 chuỗi:

BANANA@
 ANANA@
 NANA@
 ANA@
 NA@
 A@
 @

Trie hậu tố của chuỗi BANANA@ có thể biểu diễn như trong Hình 1-8



Hình 1-4. Trie hậu tố

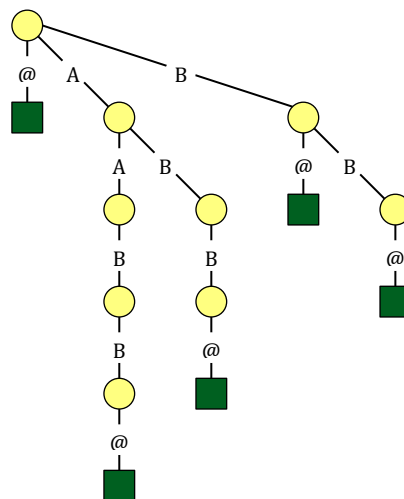
Có thể kể ra một vài ví dụ sử dụng trie hậu tố của chuỗi T :

Để kiểm tra chuỗi $Q = q[0 \dots m)$ có phải là chuỗi con của chuỗi T hay không, ta xét gốc trie và xét lần lượt các ký tự trong Q : Mỗi khi xét qua ký tự q_i thì rẽ sang nhánh con theo cạnh có nhãn là q_i . Nếu tại một bước nào đó việc chuyển xuống nhánh con thất bại do không tìm

được cạnh có nhãn tương ứng thì Q không là xâu con của T , ngược lại nếu quá trình di chuyển kết thúc ở một nút u nào đó trên trie thì Q là xâu con của T và số nút lá trong nhánh trie gốc u chính là số lần xâu Q xuất hiện trong xâu T . Ví dụ nếu $T = \text{BANANA@}$ và $Q = \text{AN}$, xét trie trong Hình 1-4, từ gốc ta di chuyển theo cạnh A rồi sau đó theo cạnh N, để dừng lại ở một nút nhánh. Trong nhánh con này có 2 nút lá (ứng với hai hậu tố) ANA@ và ANANA@ vì vậy xâu AN xuất hiện trong xâu BANANA@ đúng hai lần. Tính đúng đắn của thuật toán có thể suy ra một cách trực tiếp: xâu con của một xâu T là tiền tố của một hậu tố của T . Điều đáng chú ý ở đây là khi đã có trie hậu tố biểu diễn T , thời gian thực hiện giải thuật kiểm tra Q có phải xâu con của T hay không là $O(|Q|)$, không phụ thuộc vào chiều dài xâu T . Điều này thực sự hữu ích khi ta liên tục phải tìm kiếm những chỉ mục từ khác nhau trong một văn bản T có thể rất dài.

Một ví dụ khác sử dụng trie hậu tố là tìm xâu con lặp dài nhất (xâu con xuất hiện trong T nhiều hơn 1 lần), việc này được thực hiện rất đơn giản trên trie: Tìm nút v sâu nhất mà nhánh cây gốc v có ít nhất 2 lá, khi đó \bar{v} chính là xâu con lặp dài nhất. Như ví dụ ở Hình 1-4 **Error! Reference source not found.**, nút nhánh ở sâu nhất chính là nút có nhãn ANA . Có thể mở rộng tìm xâu con lặp bậc k dài nhất (xâu con xuất hiện trong T ít nhất k lần): Thuật toán chỉ đơn giản là tìm nút v sâu nhất mà nhánh cây gốc v có ít nhất k lá, chẳng hạn với xâu $T = \text{BANANA@}$ trong **Error! Reference source not found.** thì xâu lặp bậc 2 dài nhất là ANA , xâu lặp bậc 3 dài nhất là A .

Mặc dù trie hậu tố hỗ trợ khá nhiều phép toán hiệu quả trên xâu, việc xây dựng trie hậu tố tỏ ra khá tốn thời gian và bộ nhớ: Trong trường hợp xấu nhất, việc xây dựng trie hậu tố của xâu T cần cấp phát $\Omega(|T|^2)$ nút và mất thời gian $\Omega(|T|^2)$. Có thể lấy ví dụ về trie hậu tố của xâu $T = A^n B^n @$ (xâu gồm n ký tự A, tiếp theo là n ký tự B và kết thúc bởi ký tự đặc biệt $@$), trie này có $n^2 + 4n + 2$ nút (Hình 1-8).



Hình 1-5. Trie hậu tố của xâu AABB@ có 14 nút

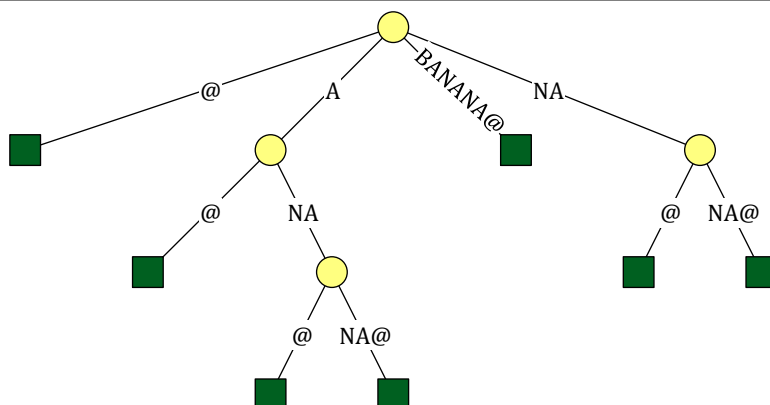
1.2. Cây hậu tố

Cây hậu tố (suffix trees) cũng là một cấu trúc dữ liệu biểu diễn các hậu tố của một xâu khác rỗng với cơ chế tương tự như trie. Cây hậu tố được tạo thành từ trie hậu tố bằng cách

chập các nút con đơn nhánh liên tiếp lại thành một nút con duy nhất và lấy dãy các nhãn cạnh bị chập thành một xâu biểu diễn nhãn cạnh.

Chính xác hơn, cây hậu tố của một xâu $T \in \Sigma^+$, ký hiệu $ST(T)$ là một cấu trúc dữ liệu dạng cây có các tính chất sau:

- ✿ Mỗi cạnh của cây có nhãn là một xâu $\in \Sigma^+$. Các cạnh đi từ một nút xuống các nút con của nó phải mang nhãn là các xâu có ký tự đầu tiên hoàn toàn phân biệt.
- ✿ Mỗi nút v trên cây hậu tố cũng mang một nhãn, nhãn của nút v , ký hiệu \bar{v} là xâu tạo thành bằng cách nối tiếp các nhãn cạnh trên đường đi từ gốc xuống nút v . Chiều dài của xâu \bar{v} : $|\bar{v}|$ được gọi là **độ sâu** của nút v , ký hiệu $depth(v)$. Theo cấu trúc của cây hậu tố, hai nút khác nhau phải có xâu nhãn khác nhau.
- ✿ Ngoại trừ nút gốc, không nút nào trên cây hậu tố có 1 nút con.
- ✿ Có tương ứng 1-1 giữa các hậu tố của T với các nút lá trên $ST(T)$: $ST(T)$ có đúng $|T|$ lá và mỗi lá có nhãn là một hậu tố của T .



Hình 1-6. Cây hậu tố

Hình 1-8 là ví dụ về cây hậu tố của xâu BANANA@.

Bổ đề 2

Cây hậu tố của một xâu độ dài n có không quá $2n$ nút

Chứng minh

Giả sử cây hậu tố có m nút nhánh tức là có tổng cộng $m + n$ nút. Ta biết rằng số cạnh của cây bằng $m + n - 1$ và tổng số con của tất cả các nút trên cây đúng bằng số cạnh trên cây. Các nút lá trên cây có số con bằng 0, các nút nhánh ngoại trừ nút gốc có số con ≥ 2 , nút gốc có ít nhất 1 con. Vì vậy tổng số con của tất cả các nút trên cây không thể nhỏ hơn $2m - 1$, từ đó ta có $2m - 1 \leq m + n - 1$ hay $m \leq n$, tức là cây hậu tố có không quá n nút nhánh và tổng cộng có không quá $2n$ nút.

Mặc dù cây hậu tố được phát kiến từ rất sớm, những nghiên cứu trên cây hậu tố lại được xây dựng từ nhiều nghiên cứu độc lập trong những lĩnh vực khác nhau với cấu trúc cây có một số khác biệt. Khi tìm kiếm những tài liệu liên quan tới cây hậu tố, ta có thể bắt gặp nhiều tên gọi khác nhau như “suffix trees”, “compacted bi-trees”, “prefix trees”, “PAT trees”, “position trees”, “repetition finder”, “subword trees”,...

Cây hậu tố được giới thiệu lần đầu tiên bởi Morrison với tên gọi cây PATRICIA [2]. Tuy vậy Weiner mới là người chuẩn hóa cấu trúc cây hậu tố (dưới tên gọi compacted bi-trees)

và đưa ra thuật toán tuyến tính xây dựng cây [3], thuật toán này sau đó được Donald Knuth bình chọn là “thuật toán của năm 1973”. Tiếp theo nghiên cứu của Weiner, các thuật toán tuyến tính xây dựng cây hậu tố liên tục cải tiến và đơn giản hóa, chẳng hạn như thuật toán của McCreight năm 1976 [4], của Slissenko năm 1983 [5]. Những thuật toán tuyến tính đầu tiên có thể làm việc trực tuyến được đề xuất bởi Kosaraju năm 1994 [6] và Ukkonen năm 1995 [7], các thuật toán này có khả năng xây dựng cây hậu tố bằng cách đọc từng ký tự trong xâu nguồn từ trái qua phải, chính vì vậy nó thích hợp khi muốn xây dựng cây hậu tố bằng cách nhận từng tín hiệu trên đường truyền.

Bên cạnh việc đề xuất các thuật toán xây dựng cây, các nghiên cứu cũng đã đưa ra thêm rất nhiều ứng dụng của cây hậu tố, đặc biệt trong lĩnh vực xử lý văn bản và dữ liệu sinh học. Tuy nhiên, có hai nhược điểm chung của các phương pháp xây dựng cây hậu tố trực tiếp, đó là:

- ✿ Các thuật toán có thể thực hiện trong thời gian tuyến tính và sử dụng bộ nhớ tuyến tính, nhưng có một hằng số lớn ẩn trong ký pháp O đánh giá độ phức tạp tính toán. Trên thực tế chương trình cài đặt thuật toán xây dựng cây hậu tố khá chậm và tốn bộ nhớ.
- ✿ Mặc dù đã có nhiều cố gắng để đơn giản hóa thuật toán nhưng cho tới nay, các mô hình cài đặt những thuật toán trên vẫn còn rất phức tạp và dễ nhầm lẫn.

1.3. Mảng hậu tố

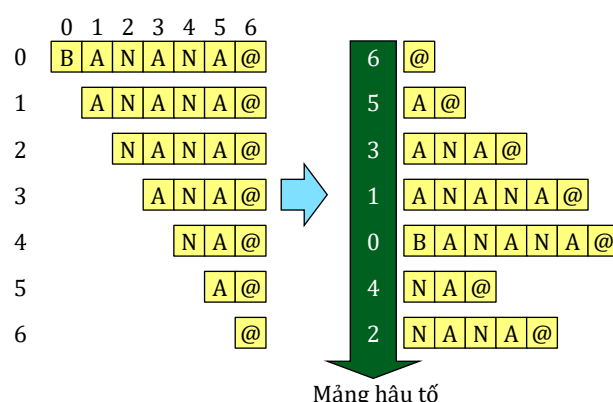
Cho một xâu $T = t[0 \dots n] \in \Sigma^+$, quy ước có duy nhất $t_{n-1} = @$. *Mảng hậu tố (suffix array)* của T , ký hiệu $SA(T)$ là thứ tự từ điển của tất cả các hậu tố của T .

Mỗi hậu tố $t[i \dots n]$ có thể được đồng nhất với vị trí i , khi đó mảng hậu tố của xâu T có thể biểu diễn như là một hoán vị $(sa_0, sa_1, \dots, sa_{n-1})$ của dãy số $(0, 1, \dots, n-1)$ sao cho:

$$t[a_0 \dots n] < t[a_1 \dots n] < \dots < t[a_{n-1} \dots n]$$

(Ở đây ta dùng ký hiệu “ $<$ ” cho quan hệ “nhỏ hơn” theo thứ tự từ điển)

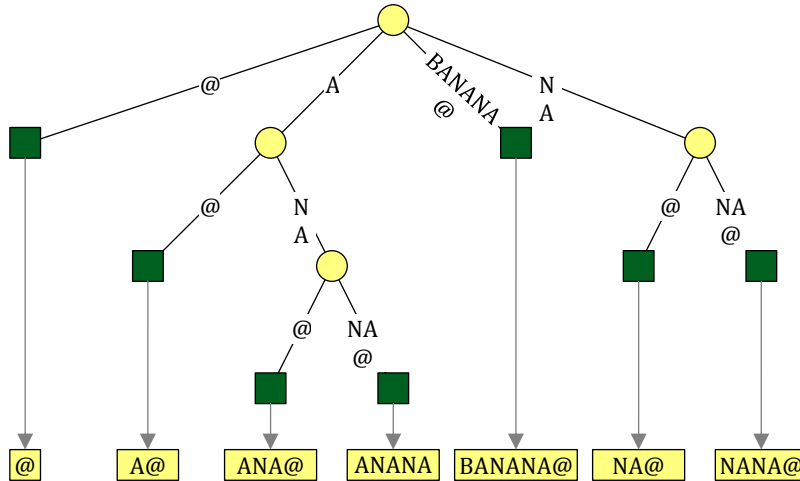
Ví dụ với xâu $T = \text{BANANA@}$, các hậu tố của T và mảng hậu tố $(7, 6, 4, 2, 1, 5, 3)$ tương ứng được chỉ ra trong Hình 1-8.



Hình 1-7. Mảng hậu tố

Mảng hậu tố $SA(T)$ của xâu T độ dài n có thể xây dựng trực tiếp từ cây hậu tố $ST(T)$ trong thời gian $O(n)$. Thuật toán có thể mô tả như sau:

- ✿ Quy định thứ tự các nút con của một nút: Theo thứ tự từ điển, nút con ứng với cạnh mang nhãn nhỏ hơn sẽ đứng trước nút con ứng với cạnh mang nhãn lớn hơn. (Thứ tự giữa hai cạnh có thể phân biệt bằng ký tự đầu tiên của nhãn cạnh)
- ✿ Duyệt cây bằng DFS bắt đầu từ gốc, khi thăm tới một nút ta lần lượt thăm các nút con của nó theo thứ tự đã quy định. Khi đó danh sách các nút lá theo thứ tự thăm sẽ ứng với danh sách các hậu tố liệt kê theo thứ tự từ điển.



Hình 1-8. Cây hậu tố và mảng hậu tố

Mảng hậu tố được đề xuất bởi Manber và Myer [8] như một sự thay thế cho cây hậu tố trong một số bài toán xử lý xâu. Ưu điểm chính của mảng hậu tố là tính đơn giản trong cấu trúc và sự tiết kiệm bộ nhớ trong biểu diễn. Manber và Mayer cũng đề xuất thuật toán $O(n \log n)$ xây dựng mảng hậu tố trực tiếp mà không phải sử dụng cây hậu tố gọi là thuật toán nhân đôi tiền tố (doubling prefix).

Cũng đã có rất nhiều nỗ lực tìm kiếm thuật toán hiệu quả hơn xây dựng mảng hậu tố. Năm 2003, hai nghiên cứu độc lập của Kärkkäinen [9] và Ko [10] đã tìm ra được hai thuật toán tuyến tính xây dựng mảng hậu tố. Một điểm đáng chú ý trong các thuật toán của Kärkkäinen và Ko là chúng đều dựa trên những nhận định rất tinh tế về tính chất của các hậu tố và mối quan hệ giữa các vị trí trong xâu. Việc phá bỏ được rào cản $\Omega(n \log n)$ trong trường hợp xấu nhất đã mở ra một tiềm năng mới cho việc sử dụng mảng hậu tố mà các nghiên cứu tiếp sau đã sử dụng để xây dựng cây hậu tố từ mảng hậu tố mà không làm tăng độ phức tạp tính toán của giải thuật.

1.4. Mảng tiền tố chung dài nhất

Tiền tố chung dài nhất (*longest common prefix*) của hai xâu x, y là xâu z có độ dài lớn nhất thỏa mãn: z vừa là tiền tố của x vừa là tiền tố của y . Ví dụ tiền tố chung dài nhất của SUFFIXTRIE và SUFFIXTREE là xâu SUFFIXTR.

Cho $T = t[0 \dots n] \in \Sigma^+$, $SA(T) = sa[0 \dots n]$ là mảng hậu tố của T . Mảng tiền tố chung dài nhất $LCP(T)$ là dãy số nguyên $lcp[0 \dots n]$ định nghĩa như sau:

- ✿ $lcp[0] = 0$;
- ✿ $\forall i > 0$: $lcp[i]$ là độ dài tiền tố chung dài nhất giữa hậu tố tại vị trí $sa[i]$ và hậu tố tại vị trí $sa[i - 1]$ trong xâu T

Ví dụ với xâu $T = \text{BANANA@}$, mảng hậu tố của T là $(0,5,3,1,0,4,2)$, ta có:

$lcp_0 = 0$;

$lcp_1 = 0$ (độ dài tiền tố chung dài nhất của $A@$ và $@$)

$lcp_2 = 1$ (độ dài tiền tố chung dài nhất của $\underline{A}NA@$ và $\underline{A}@$)

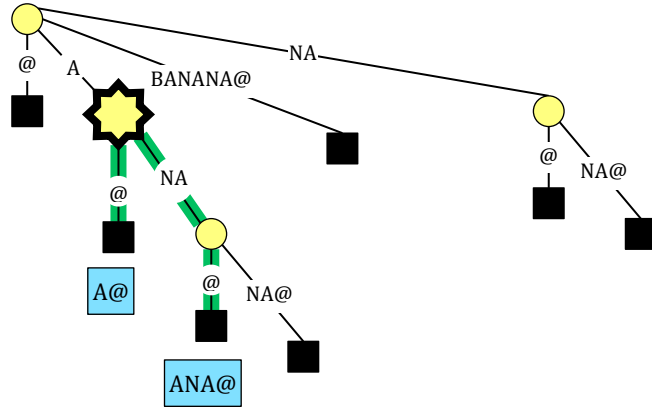
$lcp_3 = 3$ (độ dài tiền tố chung dài nhất của $\underline{ANANA@}$ và $\underline{ANA}@$)

$lcp_4 = 0$ (độ dài tiền tố chung dài nhất của $BANANA@$ và $ANANA@$)

$lcp_5 = 0$ (độ dài tiền tố chung dài nhất của $NA@$ và $BANANA@$)

$lcp_6 = 2$ (độ dài tiền tố chung dài nhất của $\underline{NANA@}$ và $\underline{NA}@$)

Bản chất của mảng tiền tố chung dài nhất có thể phân tích trên cấu trúc của cây hậu tố. Với hai hậu tố của xâu T ứng với hai lá trong $ST(T)$, tiền tố chung dài nhất của hai hậu tố là nhân của nút tiền bối chung thấp nhất (*lowest common ancestor-LCA*) của hai nút lá đó. Hình 9 là ví dụ về tiền tố chung dài nhất của hai hậu tố $A@$ và $ANA@$, tiền tố này ứng với nút mang nhãn A.



Hình 9. Tiền tố chung dài nhất giữa hai hậu tố là nhân của nút tiền bối chung thấp nhất.

Trên cây hậu tố, có rất nhiều thuật toán LCA có thể áp dụng để tìm nút tiền bối chung gần nhất của hai nút, mỗi truy vấn LCA được thực hiện trong thời gian $O(1)$ và vì thế có thể xây dựng mảng tiền tố chung dài nhất trong thời gian $O(n)$. Mặc dù vậy, Kasai [11] đã đề xuất được thuật toán tuyến tính hết sức đơn giản và hiệu quả để xây dựng mảng tiền tố chung dài nhất từ mảng hậu tố.

2. Các thuật toán xây dựng cấu trúc dữ liệu

Các thuật toán trong chuyên đề này sẽ được giới thiệu theo thứ tự sau: Trước tiên là những thuật toán xây dựng mảng hậu tố $SA(T)$ tiếp theo là thuật toán xây dựng mảng tiền tố chung dài nhất $LCP(T)$, với mục đích cuối cùng là thuật toán xây dựng cây hậu tố $ST(T)$ từ $SA(T)$ và $LCP(T)$.

2.1. Xây dựng mảng hậu tố

Bài toán: Cho xâu $T = t_0 t_1 \dots t_{n-1} \in \Sigma^+$ trong đó duy nhất $t_{n-1} = @$. Cần xây dựng mảng hậu tố $SA(T) = (a_0, a_1, \dots, a_{n-1})$:

$$t[a_0 \dots n-1] < t[a_1 \dots n-1] < \dots < t[a_{n-1} \dots n-1]$$

Mặc dù mảng hậu tố đơn giản là thứ tự từ điển của các hậu tố, việc áp dụng các thuật toán sắp xếp dựa trên phép so sánh xâu không phải là một phương pháp hay. Lý do chính là thời gian thực hiện phép so sánh hai xâu theo thứ tự từ điển sẽ tỉ lệ thuận với chiều dài xâu trong trường hợp xấu nhất. Hầu hết các thuật toán hiệu quả đều phải thực hiện trên chỉ số và chỉ dùng các phép toán trên ký tự. Trong chuyên đề này, ta đồng nhất mỗi hậu tố với vị trí ký tự đầu tiên trong xâu T , tức là nếu $T = t_0 t_1 \dots t_{n-1}$ thì hậu tố i là $t_i t_{i+1} \dots t_{n-1}$.

2.1.1. Thuật toán nhân đôi tiền tố

Phương pháp cổ điển nhất để xây dựng trực tiếp mảng hậu tố mà không cần dựng cây hậu tố có tên là thuật toán nhân đôi tiền tố, được đề xuất bởi Manber và Myers [8]. Thuật toán này cho đến nay vẫn được sử dụng phổ biến trong các kỳ thi lập trình bởi hai lý do:

- Việc cài đặt thuật toán khá đơn giản, thích hợp với việc lập trình trong thời gian hạn chế.
- Mặc dù trong trường hợp xấu nhất, thuật toán cần thời gian $\Theta(n \log n)$ để xây dựng mảng hậu tố, nhưng trung bình thuật toán chỉ cần mất thời gian $O(n)$ để thực hiện trong trường hợp dữ liệu được phân bố ngẫu nhiên trong một bảng chữ cái lớn.

□ Ý tưởng của thuật toán như sau:

Khởi tạo: Sắp xếp các hậu tố của T theo thứ tự tăng dần của ký tự đầu tiên. Điều này tương đương với việc sắp xếp các ký tự trong T theo thứ tự tăng dần. Sau đó ta gán cho mỗi hậu tố (mỗi vị trí) một khóa số nguyên $\in [0, n - 1]$ thỏa mãn: Hai hậu tố có ký tự đầu bằng nhau phải mang khóa bằng nhau, hai hậu tố có ký tự đầu khác nhau phải mang hai khóa khác nhau và hậu tố nào có ký tự đầu nhỏ hơn phải mang khóa nhỏ hơn. Việc gán khóa số mất thời gian $O(n)$. Khóa số là đại diện cho ký tự đứng đầu của các hậu tố, tức là dãy các hậu tố xếp theo thứ tự tăng dần của khóa số cũng là dãy các hậu tố theo thứ tự tăng dần của ký tự đầu tiên.

Phần chính của thuật toán được thực hiện lặp qua nhiều pha, tại một pha, giả thiết là đã có dãy các hậu tố xếp theo thứ tự tăng dần của l ký tự đầu cùng các khóa số tương ứng với thứ tự sắp xếp, thuật toán sẽ xây dựng dãy các hậu tố xếp theo thứ tự tăng dần của $2l$ ký tự đầu và dãy khóa số mới tương ứng:

- Gọi các khóa đang gán cho các hậu tố là các khóa sơ cấp (*primary keys*), mỗi hậu tố sẽ được bổ sung một khóa nữa gọi là khóa thứ cấp (*secondary keys*). Khóa thứ cấp của một hậu tố tại vị trí i chính bằng khóa sơ cấp của hậu tố đứng sau nó l vị trí ($i + l$) hoặc bằng -1 nếu $i + l \geq n$.
- Sắp xếp lại các hậu tố theo quy tắc: Trước tiên xếp tăng dần theo khóa sơ cấp, nếu hai hậu tố có khóa sơ cấp bằng nhau thì hậu tố nào có khóa thứ cấp nhỏ hơn sẽ được xếp trước. Theo giả thiết về dãy khóa sơ cấp và cách xây dựng dãy khóa thứ cấp, ta sẽ thu được dãy các hậu tố xếp theo thứ tự tăng dần của $2l$ ký tự đầu sau khi sắp xếp (khi hai hậu tố có l ký tự đầu khớp nhau thì l ký tự sau sẽ được dùng để quyết định hậu tố nào đứng trước).

- Với các hậu tố đã sắp xếp, mỗi hậu tố sẽ được gán khóa sơ cấp mới: Hậu tố đứng đầu dãy được đánh số 0. Bắt đầu từ hậu tố thứ hai trở đi trong dãy, nếu nó có cả khóa sơ cấp và thứ cấp giống với hậu tố liền trước thì khóa sơ cấp mới của nó bằng khóa sơ cấp mới của hậu tố liền trước, nếu không thì khóa sơ cấp mới của nó bằng khóa sơ cấp mới của hậu tố liền trước cộng thêm 1. Thao tác gán khóa sơ cấp mới mất thời gian $O(n)$.

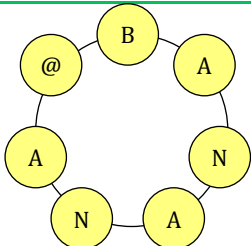
Như vậy các bước lặp lần lượt xây dựng được thứ tự của các hậu tố xếp theo 2, 4, 8, 16, 32, ... ký tự đầu tiên. Mảng hậu tố sẽ thu được sau bước lặp thứ $\lceil \lg n \rceil$. Thuật toán có thể dừng sớm tại một bước lặp nào đó mà tất cả các khóa gán cho các hậu tố là các số nguyên hoàn toàn phân biệt từ 0 tới $n - 1$ (các bước lặp sau chắc chắn không còn thay đổi thứ tự sắp xếp nữa).

Thời gian thực hiện giải thuật phụ thuộc vào thuật toán sắp xếp theo hai dãy khóa số tại mỗi bước. Có thể dùng các thuật toán sắp xếp so sánh, chẳng hạn như QuickSort. Tuy nhiên vì tập các giá trị khóa là các số nguyên nằm trong phạm vi từ 1 tới n , ta có thể áp dụng các thuật toán sắp xếp cơ số (*Radix Sort*) hoặc đếm phân phối (*Counting Sort*) để đạt tốc độ cao hơn với thời gian thực hiện giải thuật sắp xếp là $O(n)$. Từ đó thuật toán nhân đôi tiền tố có thể thực hiện trong thời gian $O(n \log n)$.

❑ Cài đặt

Để cài đặt thuật toán nhân đôi tiền tố được đơn giản và hiệu quả, ta cần đưa ra một vài nhận xét để có phương pháp tổ chức dữ liệu hợp lý.

Xét hoán vị vòng quanh $t_i t_{i+1} \dots t_{n-1} t_1 \dots t_{i-1}$ của xâu T tại vị trí i . Hoán vị vòng quanh này là ghép của hậu tố $t_{i \dots n-1}$ với tiền tố $t_{0 \dots i-1}$. Có thể coi hoán vị vòng quanh được tạo thành bằng cách viết các ký tự trong T quanh một vòng tròn theo chiều kim đồng hồ rồi lấy n ký tự liên tiếp theo chiều đã định bắt đầu từ vị trí i . Với việc sử dụng ký tự cảm canh @, thứ tự từ điển của các hậu tố của T cũng là thứ tự từ điển của các hoán vị vòng quanh tương ứng. Điều này có thể suy ra được vì ký tự @ là ký tự nhỏ nhất trong bảng chữ cái và ký tự này chỉ xuất hiện một lần ở cuối xâu T . Bảng dưới đây là thứ tự từ điển của các hậu tố cũng như hoán vị vòng quanh tại cùng vị trí.

	Hậu tố	Hoán vị vòng quanh
	@	@BANANA
	A@	A@BANAN
	ANA@	ANA@BAN
	ANANA@	ANANA@B
	BANANA@	BANANA@
	NA@	NA@BANA
	NANA@	NANA@BA

Bằng việc sử dụng hoán vị vòng quanh, khái niệm vị trí đứng trước/sau l bước so với vị trí i luôn tồn tại trên vòng tròn nên ta không cần phải xử lý trường hợp riêng khi một vị trí không tồn tại trong dãy.

Đồng nhất mỗi hoán vị vòng quanh với vị trí của ký tự đứng đầu. Giả sử đầu mỗi pha lặp, ta có dãy $(a_0, a_1, \dots, a_{n-1})$ là dãy các hoán vị vòng quanh xếp theo thứ tự từ điển của l ký tự đầu tiên. Gọi l ký tự đầu tiên của mỗi hoán vị vòng quanh là **đoạn sơ cấp** và l ký tự tiếp theo đoạn sơ cấp là **đoạn thứ cấp**.

Xây dựng dãy $B = (b_0, b_1, \dots, b_{n-1})$ như sau: b_i là vị trí đứng trước l bước so với a_i trên vòng tròn. Đây chính là dãy hoán vị vòng quanh xếp theo thứ tự từ điển của đoạn thứ cấp.

Tiếp theo, ta sắp xếp lại dãy $(b_0, b_1, \dots, b_{n-1})$ theo thứ tự tăng dần của khóa sơ cấp tương ứng bằng một **thuật toán sắp xếp ổn định** (chẳng hạn như thuật toán đếm phân phối) để được dãy $(a_0, a_1, \dots, a_{n-1})$ mới. Trong thuật toán sắp xếp ổn định, hai hoán vị vòng quanh có đoạn sơ cấp giống nhau thì hoán vị nào đang đứng trước (có đoạn thứ cấp nhỏ hơn) vẫn sẽ đứng trước. Từ đó suy ra dãy $(a_0, a_1, \dots, a_{n-1})$ mới thu được chính là thứ tự từ điển của các hoán vị vòng quanh theo $2l$ ký tự đầu tiên. Chú ý rằng ta không cần sử dụng một mảng nào để chứa khóa thứ cấp, khóa thứ cấp của mỗi vị trí được suy ra từ khóa sơ cấp của vị trí đứng sau nó l bước.

Để tiện lợi cho việc gán khóa sơ cấp, ta có thêm mảng boolean $mark[0 \dots n - 1]$ để đánh dấu. Sau mỗi pha lặp, $mark[i]$ sẽ được đặt bằng True nếu a_i phải mang khóa số khác với a_{i-1} . Nếu trước pha lặp $mark[i] = \text{True}$ tức là a_i có khóa sơ cấp khác với a_{i-1} thì sau bước lặp đó a_i vẫn mang khóa sơ cấp khác a_{i-1} , ta chỉ đặt thêm $mark[i] = \text{True}$ nếu sau pha lặp a_i mang khóa thứ cấp khác với a_{i-1} ;

Dưới đây là đoạn chương trình tính mảng hậu tố theo thuật toán nhân đôi tiền tố

Input:

Dòng 1: Độ dài xâu T ($n \leq 10^5$)

Dòng 2: Xâu T (có ký tự cần canh @ đứng cuối)

Output

Mảng hậu tố của T

Sample Input	Sample Output
7 BANANA@	6 5 3 1 0 4 2

SUFFIXARRAY.PAS ✓ Tính mảng hậu tố

```
{ $MODE OBJFPC }
program SuffixArrayConstruction;
const
    maxN = 100000;
type
    TAlphabet = '@'..'Z';
var
    T: array [0..maxN - 1] of TAlphabet;
    n: Integer;
    key, head, a, b: array [0..maxN - 1] of Integer;
    mark: array [0..maxN - 1] of Boolean;

procedure Init; // Khởi tạo
var
    i: Integer;
    c: TAlphabet;
    ccount: array [TAlphabet] of Integer;
begin
    ReadLn(n);
    for i := 0 to n - 1 do Read(T[i]);
    // Thuật toán đếm phân phối, tạo dãy a[0..n-1] là các hoán vị vòng quanh xếp theo chữ cái đầu
```

```

FillChar(ccount, SizeOf(ccount), 0);
for i := 0 to n - 1 do Inc(ccount[T[i]]);
for c := Succ(Low(TAlphabet)) to High(TAlphabet) do
  Inc(ccount[c], ccount[pred(c)]);
for i := n - 1 downto 0 do
  begin
    c := T[i];
    Dec(ccount[c]);
    a[ccount[c]] := i;
  end;
//Khởi tạo mảng mark[0...n-1]
mark[0] := True;
for i := 1 to n - 1 do
  mark[i] := T[a[i]] <> T[a[i - 1]]; //a[i] phải mang khóa khác a[i-1]
end;

procedure SuffixArray; //Thuật toán nhân đôi tiền tố
var
  i, j, nkeys, kv: Integer;
  len: Integer;
begin
  len := 1;
  while len < n do
    begin
      //Trước mỗi pha lặp đã có a[0...n-1] là dãy các HVVQ xếp theo đoạn sơ cấp gồm k ký tự đầu
      //Dựa vào mảng mark tính các giá trị khóa sơ cấp
      //và tạo b[0...n-1] là dãy các HVVQ xếp theo đoạn thứ cấp
      nkeys := 0;
      for i := 0 to n - 1 do
        begin
          if mark[i] then //Tính luôn head[nkeys] = i là vị trí đầu tiên sẽ mang khóa sơ cấp nkeys
            begin
              head[nkeys] := i;
              Inc(nkeys);
            end;
          key[a[i]] := nkeys - 1;
          b[i] := (a[i] - len + n) mod n; //b[i] = hvvq đứng trước vị trí a[i] đúng len bước
        end;
      if nkeys = n then Break; //Nếu các khóa sơ cấp hoàn toàn phân biệt thì xong
      for i := 0 to n - 1 do
        begin
          kv := key[b[i]];
          a[head[kv]] := b[i];
          Inc(head[kv]);
        end;
      //a[0...n-1] giờ là dãy hvvq xếp theo 2len ký tự đầu, cập nhật lại mảng mark
      kv := -1;
      for i := 0 to n - 1 do
        begin
          j := (a[i] + len) mod n; //j đứng sau a[i] đúng len bước, key[j] là khóa thứ cấp của a[i]
          if key[j] <> kv then //key[j] = khóa thứ cấp của a[i] khác với khóa thứ cấp của a[i-1]
            begin
              mark[i] := True; //Đặt mark để đánh dấu
              kv := key[j];
            end;
        end;
      len := len shl 1; //Nhân đôi độ dài tiền tố
    end;
end;

procedure PrintResult; //In kết quả
var

```

```

i: Integer;
begin
  for i := 0 to n - 1 do Write(a[i], ' ');
  WriteLn;
end;

begin
  Init;
  SuffixArray;
  PrintResult;
end.

```

2.1.2. Thuật toán chia để trị

Một trong những kỹ thuật quan trọng được dùng trong thuật toán nhân đôi tiền tố đó là thay thế việc so sánh chuỗi bởi phép so sánh số nguyên. Kỹ thuật này gọi là mã hóa (*encoding*) bằng khóa số.

Để xây dựng mảng hậu tố, còn có hai thuật toán khác được phát triển độc lập nhưng đều dựa trên kỹ thuật chia để trị và cơ chế mã hóa bằng khóa số. Tuy cách tiếp cận có khác nhau nhưng chúng đều là các thuật toán tuyến tính (cả về thời gian và bộ nhớ). Ngoài việc lấp đầy khoảng trống lý thuyết*, những thử nghiệm trên dữ liệu thực tế cũng cho thấy ưu điểm rõ rệt của những phương pháp này so với thuật toán nhân đôi tiền tố khi xử lý dữ liệu lớn.

Thuật toán của Kärkkäinen và Sanders (2003) [9]: Tại mỗi bước, hai phần ba số hậu tố sẽ được mã hóa theo 3 ký tự đầu và được sắp xếp bằng đệ quy, một phần ba còn lại được sắp dựa vào thứ tự từ điển của số hậu tố đã sắp sau đó trộn kết quả lại. Kích thước dữ liệu giảm xuống còn $2/3$ sau mỗi bước lặp. Thời gian thực hiện giải thuật là $T(n) = T\left(\frac{2n}{3}\right) + O(n) = O(n)$.

Thuật toán của Ko và Aluru (2003) [10]: Tại mỗi bước, các hậu tố được phân làm hai loại: Loại S là các hậu tố tại vị trí i mà nhỏ hơn hậu tố tại vị trí $i + 1$ và loại L là các hậu tố tại vị trí i mà lớn hơn hậu tố tại vị trí $i + 1$. Các tác giả chứng minh được rằng chỉ cần sắp xếp được một trong hai loại thì chỉ cần một thuật toán tuyến tính là có thể trộn các hậu tố còn lại vào thành toàn bộ mảng hậu tố. Từ đó bài toán sắp xếp n hậu tố quy về bài toán sắp xếp tối đa $\left\lfloor \frac{n}{2} \right\rfloor$ hậu tố. Thời gian thực hiện giải thuật là $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n) = O(n)$.

Khuôn khổ của chuyên đề không cho phép chúng tôi trình bày toàn bộ chứng minh và mã nguồn của hai thuật toán này. Bạn đọc có thể tham khảo bài báo gốc về chi tiết của chúng. Chúng ta thừa nhận đã có thuật toán xây dựng mảng hậu tố trong thời gian và bộ nhớ tuyến tính.

* Vì thuật toán dựng cây hậu tố trực tiếp là thuật toán tuyến tính nhưng khá phức tạp, người ta đã tìm được thuật toán tuyến tính đơn giản hơn nhưng cần dựng cây hậu tố một cách gián tiếp qua mảng hậu tố. Tuy vậy điều này sẽ không còn ý nghĩa lý thuyết nếu như thuật toán xây dựng mảng hậu tố không phải là thuật toán tuyến tính.

2.2. Xây dựng mảng tiền tố chung dài nhất.

Bài toán: Cho chuỗi $T = t_0 t_1 \dots t_{n-1} \in \Sigma^+$ trong đó duy nhất $t_{n-1} = @$ cùng với mảng hậu tố tương ứng $SA(T) = (a_0, a_1, \dots, a_{n-1})$. Cần phải xây dựng mảng tiền tố chung dài nhất $LCP(T) = (l_0, l_1, \dots, l_{n-1})$:

- $l_0 = 0$,
- $\forall i > 0, lcp_i$ độ dài tiền tố chung dài nhất giữa hậu tố a_i và hậu tố a_{i-1} .

2.2.1. Thuật toán

Thuật toán phổ biến nhất để xây dựng mảng $LCP(T)$ được đề xuất bởi Kasai [11]. Sau đây ta trình bày thuật toán đó.

Trước hết ta tính mảng $rank_{0..n-1}$ trong đó $rank_i$ là vị trí của hậu tố i trong mảng hậu tố. Tức là:

$$rank_i = j \Leftrightarrow a_j = i$$

Mảng $(l_0, l_1, \dots, l_{n-1})$ sẽ được xây dựng theo thứ tự: $l[rank_1], l[rank_2], \dots, l[rank_n]$, trong đó và $l[rank_{i+1}]$ sẽ được tính dựa vào $l[rank_i]$.

Với mỗi giá trị i , gọi $q = l[rank_i]$, và $j = a[rank_i - 1]$ là hậu tố đứng liền trước hậu tố i trong mảng hậu tố. Theo định nghĩa về mảng $LCP(T)$ ta có q là độ dài tiền tố chung dài nhất giữa hai hậu tố i và j . Loại bỏ ký tự đầu tiên của cả hai hậu tố này, ta có: tiền tố chung dài nhất giữa hai hậu tố $i + 1$ và $j + 1$ có độ dài $q - 1$ nếu $q \geq 1$ và bằng 0 trong trường hợp ngược lại.

Nếu $q \geq 1$, ta có $t_i = t_j$. Mặt khác, hậu tố j đứng liền trước hậu tố i trong mảng hậu tố nên nếu loại bỏ ký tự đầu giống nhau từ hai hậu tố này thì thu được hai hậu tố mới $i + 1$ và $j + 1$ trong đó hậu tố $j + 1$ vẫn nhỏ hơn và đứng trước hậu tố $i + 1$ trong mảng hậu tố. Xét trên thứ tự từ điển, hậu tố $j + 1$ có thể không đứng liền trước hậu tố $i + 1$ nhưng bởi tiền tố chung dài nhất của chúng có độ dài $q - 1$, mọi hậu tố nằm giữa chúng theo thứ tự từ điển đều phải có $q - 1$ ký tự đầu trùng với hậu tố $j + 1$ cũng như với hậu tố $i + 1$. Điều này chỉ ra rằng hậu tố $i + 1$ có ít nhất $q - 1$ ký tự đầu trùng với hậu tố đứng liền trước nó theo thứ tự từ điển hay $l[rank_{i+1}] \geq q - 1$. Dĩ nhiên bất đẳng thức này đúng cả với trường hợp $q = 0$. Từ đó ta có bổ đề sau:

Bổ đề 3

Với $\forall i: 0 \leq i < n - 1$, ta có $l[rank_{i+1}] \geq l[rank_i] - 1$.

Bổ đề 3 cho phép ta cài đặt thuật toán tính mảng tiền tố chung dài nhất bằng một thuật toán rất ngắn gọn:

```
l[0] := 0;
q := 0;
for i := 0 to n - 2 do
  begin //l[i] ≥ q. Chú ý rank[i] luôn > 0 do rank[n - 1] luôn bằng 0 (hậu tố @)
    j := a[rank[i] - 1]; //j là hậu tố đứng liền trước i trong mảng hậu tố
    while t[i + q] = t[j + q] do q := q + 1; //Tăng q nếu ký tự thứ q+1 của hậu tố i và hậu tố j khớp nhau
    l[rank[i]] := q; //Do ký tự thứ q + 1 của hậu tố i và hậu tố j khác nhau
    if q > 0 then q := q - 1; //Giảm q chuẩn bị cho bước sau
  end;
```

Vì T có duy nhất một ký tự cầm canh ở cuối nhỏ hơn mọi ký tự khác, hậu tố $n - 1$ của T chắc chắn đứng đầu thứ tự từ điển trong các hậu tố và như vậy $l[rank_{n-1}] = l[0] = 0$ và ta không cần tính $l[rank_{n-1}]$ nữa. Mỗi bước lặp, thuật toán tính $l[rank_i]$ với một điều kiện chắc chắn là $l[rank_i] \geq q$. Gọi j là hậu tố đứng liền trước hậu tố i trong thứ tự từ điển của mảng hậu tố, khi đó hai hậu tố này có ít nhất q ký tự đầu tiên trùng nhau. Tăng q lên đến khi ký tự thứ $q + 1$ của hai hậu tố i và j không khớp, ta có $l[rank_i] = q$. Việc cuối cùng là giảm q đi 1 đơn vị (nếu $q > 0$) để chuẩn bị cho bước sau (tính $l[rank_{i+1}]$).

Thời gian thực hiện giải thuật có thể đánh giá qua số lần giá trị q tăng hoặc giảm bởi hai lệnh $q := q - 1$ và $q := q + 1$. Giá trị của q được khởi tạo bằng 0. Vòng lặp while chắc chắn tìm được giá trị q mà ký tự thứ $q + 1$ của hai hậu tố t_i và t_j không khớp nhau ($t[i + q] \neq t[j + q]$), do đó những lệnh $q := q + 1$ không thể làm cho q vượt quá $n - 1$. Ngoài ra có tối đa $n - 1$ lần q bị giảm (bởi lệnh $q := q - 1$) sau mỗi bước lặp, suy ra lệnh tăng q ($q := q + 1$) có số lần thực hiện không vượt quá $2n - 2$. Vậy thuật toán Kasai có thể dựng được mảng $LCP(T)$ trong thời gian $O(n)$.

2.2.2. Cài đặt

Input:

- Dòng 1: Độ dài xâu T ($n \leq 10^5$)
- Dòng 2: Xâu T (có ký tự cầm canh @ đứng cuối).
- Dòng 3: Các giá trị a_0, a_1, \dots, a_{n-1} ứng với mảng hậu tố $SA(T)$

Output

Mảng tiền tố chung dài nhất $LCP(T)$

Sample Input	Sample Output
7 BANANA@ 6 5 3 1 0 4 2	0 0 1 3 0 0 2

LCPARRAY.PAS ✓ Tính mảng tiền tố chung dài nhất

```
{ $MODE OBJFPC }
program LCPArrayConstruction;
const
  maxN = 100000;
type
  TAlphabet = '@'..'Z';
var
  t: array[0..maxN - 1] of TAlphabet;
  a, rank, l: array[0..maxN - 1] of Integer;
  n: Integer;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 0 to n - 1 do Read(t[i]);
  ReadLn;
  for i := 0 to n - 1 do Read(a[i]);
end;
```



```

procedure LCPArray; //Tính mảng tiền tố chung dài nhất
var
  i, j, q: Integer;
begin
  for i := 0 to n - 1 do rank[a[i]] := i; //Tính hạng của mỗi hậu tố
  l[0] := 0;
  q := 0;
  for i := 0 to n - 2 do
    begin
      j := a[rank[i] - 1]; //j là hậu tố đứng liền trước i trong thứ tự từ điển của mảng hậu tố
      while t[i + q] = t[j + q] do Inc(q); //Tăng q nếu ký tự thứ q+1 của hậu tố i và hậu tố j khớp
    nhau
      l[rank[i]] := q; //Do ký tự thứ q + 1 của hậu tố i và hậu tố j khác nhau
      if q > 0 then Dec(q); //Giảm q chuẩn bị cho bước sau: tính l[rank[i + 1]]
    end;
  end;

procedure PrintResult;
var
  i: Integer;
begin
  for i := 0 to n - 1 do Write(l[i], ' ');
  WriteLn;
end;

begin
  Enter;
  LCPArray;
  PrintResult;
end.

```

2.3. Xây dựng cây hậu tố

Bài toán: Cho xâu $T = t_0 t_1 \dots t_{n-1} \in \Sigma^+$ trong đó có duy nhất $t_{n-1} = @$. Cho mảng hậu tố $SA(T) = (a_0, a_1, \dots, a_{n-1})$ và mảng tiền tố chung dài nhất $LCP(T) = (l_0, l_1, \dots, l_{n-1})$ của T . Cần xây dựng cây hậu tố của T : $ST(T)$.

2.3.1. Thuật toán

Ý tưởng của thuật toán là chèn lần lượt các hậu tố trong mảng hậu tố vào cây. Bắt đầu từ một cây chỉ gồm nút gốc, thuật toán lần lượt xét các hậu tố theo thứ tự từ điển. Mỗi khi chèn một hậu tố, có thể có một nút nhánh và một nút lá được bổ sung vào cây, trong đó nút lá có nhãn là hậu tố vừa chèn.

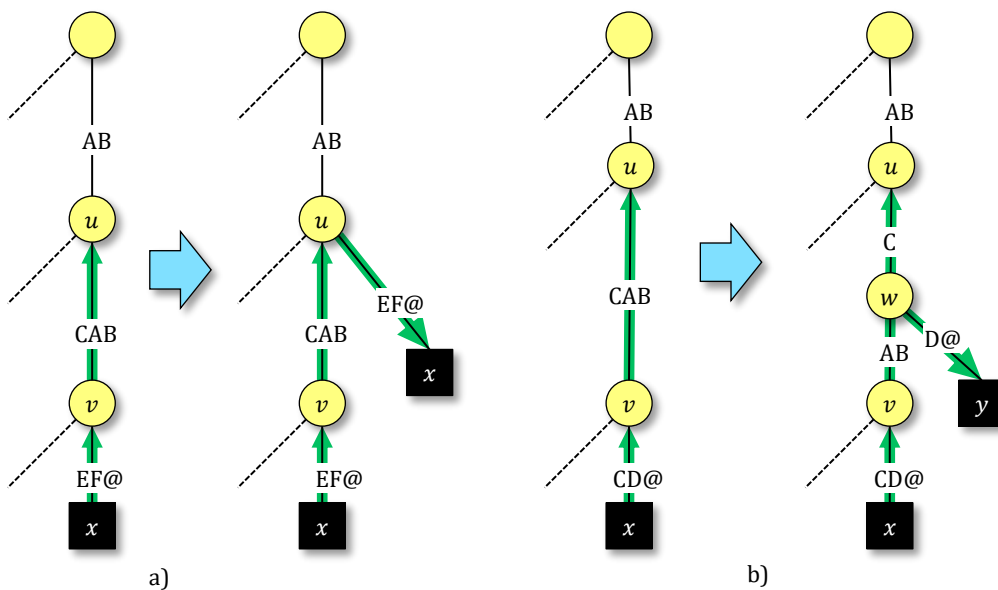
Cụ thể hơn, ta bắt đầu từ một cây chỉ gồm nút gốc và nút lá cực phải x được đặt bằng gốc. Vị trí nút x sẽ được cập nhật bằng lá mới được chèn vào cây tại mỗi bước. Xét lần lượt các hậu tố trong mảng hậu tố. Mỗi khi xét tới hậu tố a_i , ta đi từ lá cực phải x lên phía gốc cho tới khi gặp một nút u có độ sâu $depth(u) \leq l_i$. Gọi v là nút con của u đi qua trong quá trình di chuyển từ x lên u .

Trường hợp 1: Nếu $depth(u) = l_i$, ta chỉ cần tạo một nút lá mới y làm con của u và đặt nhãn cạnh (u, y) sao cho nhãn cạnh này nối với \bar{u} chính là hậu tố a_i (nhãn cạnh này bằng $t[a_i + depth(u) \dots n - 1]$). Hình 10 a) là ví dụ về cây vừa được chèn ABCABEF@ và sau đó chèn thêm ABEF@, trong ví dụ này ta tìm được nút u có $\bar{u} = AB$ là tiền tố chung dài nhất của ABCABEF@ và ABEF@

Trường hợp 2: Nếu $depth(u) < lcp_i$:

- Thêm nút w làm con u và cha v để tách cạnh (u, v) làm hai cạnh (u, w) và (w, v) . Nhãn cạnh (u, v) cũng được tách ra hai phần: $l_i - depth(u)$ ký tự đầu được lấy làm nhãn cạnh (u, w) , các ký tự sau được lấy làm nhãn cạnh (w, v)
- Nút w có độ sâu đúng bằng lcp_i , đặt $u := w$, việc chèn quy về trường hợp 1.

Hình 10 b) là cũng là ví dụ về cây vừa được chèn ABCABCD@ và sau đó chèn thêm ABCD@, quá trình đi ngược từ lá x lên dừng ở nút u nhưng $\bar{u} = AB$ chưa phải là tiền tố chung dài nhất của ABCDE@ và ABCFG@ (bằng ABC). Ta cần “chia” cạnh (u, v) ra làm hai phần bởi nút mới w sao cho $\bar{w} = ABC$ sau đó bổ sung nút lá y làm con w với nhãn cạnh (w, y) là D@ để có được $\bar{y} = ABCD@$.



Hình 10. Chèn ABCGH@ vào cây hậu tố

Không khó khăn để chứng minh giải thuật dựng cây hậu tố từ mảng hậu tố và mảng tiền tố chung dài nhất như trên có thể thực hiện trong thời gian $O(n)$ vì đó đơn thuần là phép duyệt các cạnh trên cây:

- Mỗi cạnh được duyệt qua tối đa một lần trong phép di chuyển từ lá lên gốc (cạnh đi qua cuối cùng có thể bị xóa và thay bằng 2 cạnh mới)
- Mỗi phép chèn hậu tố vào cây làm tăng số cạnh lên nhiều nhất là 2.

2.3.2. Cài đặt

Input:

- Dòng 1: Độ dài xâu $T (n \leq 10^5)$
- Dòng 2: Xâu T (có ký tự cảm canh @ đứng cuối).
- Dòng 3: Các giá trị a_0, a_1, \dots, a_{n-1} ứng với mảng hậu tố $SA(T)$
- Dòng 3: Các giá trị l_0, l_1, \dots, l_{n-1} ứng với mảng tiền tố chung dài nhất $LCP(T)$

Output

Cây hậu tố $ST(T)$

Sample Input	Sample Output
7	--ROOT
BANANA@	{
6 5 3 1 0 4 2	--@
0 0 1 3 0 0 2	--A
	{
	--@
	--NA
	{
	--@
	--NA@
	}
	}
	--BANANA@
	--NA
	{
	--@
	--NA@
	}
	}

Trong chương trình cài đặt dưới đây, ta tổ chức các nút của cây hậu tố trong một mảng *nodes*. Mỗi nút là một bản ghi gồm các trường:

- *indexL, indexH*: Nhãn của cạnh nối từ nút cha là xâu $t[indexL \dots indexH - 1]$
- *depth*: Độ sâu của nút
- *parent*: Chỉ số nút cha
- *child['@' ... 'Z']*: *child[c]* là chỉ số nút con ứng với các nhãn cạnh có ký tự đầu là *c*.

SUFFIXTREECONSTRUCTION.PAS ✓ Dựng cây hậu tố

```
{ $MODE OBJFPC }
program SuffixTreeConstruction;
const
  maxN = 100000;
type
  TAlphabet = '@'..'Z';
  TNode = record // Cấu trúc nút
    indexL, indexH: Integer; // Nhãn cạnh nối từ nút cha là t[indexL...indexH - 1]
    depth: Integer; // Độ sâu
    parent: Integer; // Chỉ số nút cha
    child: array[TAlphabet] of Integer; // Chỉ số các nút con
  end;
var
  T: array[0..maxN - 1] of TAlphabet;
  a, l: array[0..maxN - 1] of Integer;
  nodes: array[1..2 * maxN] of TNode;
  n: Integer;
  x, nodeptr: Integer;

procedure Enter; // Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n);
  for i := 0 to n - 1 do Read(T[i]);
  ReadLn;
  for i := 0 to n - 1 do Read(a[i]);
  ReadLn;
```

```

    for i := 0 to n - 1 do Read(l[i]);
end;

function NewNode: Integer; //Tạo nút mới có chỉ số nodeptr
begin
    Inc(nodeptr);
    FillChar(nodes[nodeptr], SizeOf(nodes[nodeptr]), 0);
    Result := nodeptr;
end;

procedure Init; //Tạo cây chỉ gồm nút gốc chỉ số 1
begin
    nodeptr := 0;
    x := NewNode;
end;

procedure SetLink(u, v: Integer); //Cho v làm con của u
var
    ch: TAlphabet;
begin
    ch := T[nodes[v].indexL]; //Đọc ký tự đầu nhãn cạnh
    nodes[v].parent := u;
    nodes[u].child[ch] := v;
end;

//Thủ tục chính: Chèn hậu tố suffindex vào cây, biết độ dài tiền tố chung dài nhất của hậu tố này với hậu tố vừa
chèn là lcpvalue
procedure InsertSuffix(suffindex, lcpvalue: Integer);
var
    u, v, w, y: Integer;
    k, p: Integer;
begin
    u := x;
    while nodes[u].depth > lcpvalue do //Đi từ lá x lên gốc, tìm nút u có độ sâu ≤ lcpvalue
    begin
        v := u;
        u := nodes[v].parent;
    end;
    if nodes[u].depth < lcpvalue then //Nếu u có độ sâu < lcpvalue
    begin
        //Tạo nút w có độ sâu lcpvalue chèn vào giữa cạnh (u, v), nhãn (u, w) nối với nhãn (w, v) = nhãn (u, v)
        cũ
        w := NewNode;
        k := nodes[v].indexL;
        p := lcpvalue - nodes[u].depth;
        nodes[w].indexL := k;
        nodes[w].indexH := k + p;
        nodes[w].depth := lcpvalue;
        nodes[v].indexL := k + p;
        SetLink(u, w);
        SetLink(w, v);
        u := w;
    end;
    //u có độ sâu bằng lcpvalue, tạo lá y làm con của u
    y := NewNode;
    with nodes[y] do
    begin
        indexL := suffindex + lcpvalue;
        indexH := n;
        depth := n - suffindex; //Độ sâu của y bằng chiều dài hậu tố
    end;
    SetLink(u, y);

```

```

    x := y; //Cập nhật lá cực phải mới
end;

procedure SuffixTree;
var
    i: Integer;
begin
    for i := 0 to n - 1 do
        InsertSuffix(a[i], l[i]); //Chèn lần lượt các hậu tố theo thứ tự từ điển vào cây
    end;

//Các thủ tục trình bày output, không quan trọng
    procedure WriteBlank(nb: Integer);
    var
        i: Integer;
    begin
        for i := 1 to nb do Write(' ');
    end;

    procedure Visit(i: Integer; indent: Integer);
    var
        ch: TAlphabet;
        j: Integer;
    begin
        if i = 0 then Exit;
        WriteBlank(indent);
        Write('--');
        if i = 1 then Write('ROOT')
        else
            with nodes[i] do
                for j := indexL to indexH - 1 do Write(t[j]);
            WriteLn;
            if nodes[i].indexH <> n then
                begin
                    WriteBlank(indent + 2); WriteLn('{');
                    for ch := Low(TAlphabet) to High(TAlphabet) do
                        Visit(nodes[i].child[ch], indent + 4);
                    WriteBlank(indent + 2); WriteLn('}');
                end;
        end;
    end;

    procedure PrintResult;
    begin
        Visit(1, 0);
    end;

begin
    Enter;
    Init;
    SuffixTree;
    PrintResult;
end.

```

3. Bài tập ví dụ

3.1. Mật mã ẩn (ACM 2003)

Cho xâu độ dài $n \leq 10^5$, tìm hoán vị vòng quanh có thứ tự từ điển nhỏ nhất. Ví dụ với xâu “ALABALA” thì hoán vị vòng quanh nhỏ nhất là “AALABAL”

Thuật toán:

Mặc dù có thuật toán $O(n)$ cài đặt đơn giản hơn, việc áp dụng các thuật toán dựng mảng hậu tố cũng là một giải pháp không mất nhiều công sức suy nghĩ. Chú ý là việc không được sử dụng ký tự cầm canh có thể dẫn tới vài sửa đổi nhỏ trong cài đặt thuật toán

3.2. Số xâu con phân biệt (IOI training camp 2003)

Bạn được cho xâu T độ dài không quá 10^5 , cho biết có bao nhiêu xâu con khác nhau và khác rỗng của T .

Thuật toán

Bài toán đơn thuần là đếm số nút trên trie hậu tố ngoại trừ nút gốc, tương đương với việc đếm số cạnh trên trie. Vấn đề tốn kém bộ nhớ và thời gian khi xây dựng trie có thể được khắc phục bằng cây hậu tố với một số sửa đổi nhỏ. Tuy nhiên cách hay nhất là dùng mảng tiền tố chung dài nhất.

Giả sử ta có mảng hậu tố $(a_0, a_1, \dots, a_{n-1})$ và mảng tiền tố chung dài nhất (l_0, l_1, \dots, l_n) . Chèn lần lượt các hậu tố vào trie theo thứ tự từ điển, phân tích quá trình hậu tố a_i (độ dài $n - a_i$) được chèn vào trie, l_i ký tự đầu được duyệt qua mà không có sự bổ sung nút và cạnh. Những ký tự sau, mỗi ký tự sẽ bổ sung 1 cạnh và 1 nút trên trie. Suy ra đáp số là:

$$\sum_{i=0}^{n-1} (n - a_i - l_i) = \frac{n(n+1)}{2} - \sum_{i=0}^{n-1} l_i$$

(Do $\sum_{i=0}^{n-1} (n - a_i) = \frac{n(n+1)}{2}$)

3.3. Xâu con (Training Camp 2003)

Cho xâu T gồm n ký tự ($n \leq 10^5$) và một số $k \leq n$, tìm xâu con dài nhất xuất hiện trong xâu T ít nhất k lần.

Thuật toán:

Ta đã trình bày phương pháp dùng trie hậu tố, trên cây hậu tố cũng có thể dùng phương pháp tương tự. Tuy nhiên bài toán này có thể giải một cách đơn giản bằng mảng hậu tố và mảng tiền tố chung gần nhất. Gợi ý: Điều kiện để có một xâu độ dài q xuất hiện k lần trong xâu T là trong mảng $LCP(T)$ tồn tại $k - 1$ số liên tiếp $\geq q$.

3.4. Xâu con đối xứng dài nhất (USACO training gate)

Cho xâu S độ dài $n \leq 10^5$, tìm xâu con đối xứng dài nhất.

Thuật toán:

Gọi \bar{S} là xâu đảo ngược của xâu S , nhận xét rằng một xâu đối xứng độ dài lẻ của S có ký tự đứng giữa là s_i sẽ phải có dạng $\bar{A}s_iA$ trong đó A là một xâu con của S bắt đầu từ vị trí $i + 1$ và cũng là xâu con của S' bắt đầu tại vị trí $n - i - 1$. Dựng xâu $T = S + \bar{S}$ và mảng hậu tố $SA(T)$, khi đó:

- Mọi xâu con của S bắt đầu từ vị trí i phải là tiền tố của hậu tố thứ i của T .
- Mọi xâu con của \bar{S} bắt đầu từ vị trí $n - i$ phải là tiền tố của hậu tố thứ $2n - 1 - i$ của T

Vấn đề quy về tìm tiền tố chung dài nhất giữa hai hậu tố của T . Trên mảng hậu tố $SA(T) = (a_0, a_1, \dots, a_{2n-1})$ và mảng $LCP(T) = (l_0, l_1, \dots, l_{2n-1})$. Tiền tố chung dài nhất giữa hậu tố a_i và hậu tố a_j ($i < j$) là giá trị nhỏ nhất trong các giá trị $lcp[i + 1 \dots j]$, truy vấn giá trị nhỏ nhất trong một khoảng liên tiếp (*range-minimum query*) có thể được thực hiện trong thời gian $O(\log n)$ bằng cấu trúc dữ liệu segment trees hoặc thực hiện trong thời gian $O(1)$ bằng phép quy dẫn LCA hay Bucket Pointers. Khi xét trên mọi vị trí i , thuật toán tìm sâu con đối xứng dài nhất độ dài lẻ mất thời gian $O(n \log n)$ hoặc $O(n)$ tùy theo cấu trúc dữ liệu được lựa chọn.

Vấn đề tương tự trong việc tìm sâu con đối xứng dài nhất độ dài chẵn.

3.5. Mẫu ghép (Polish Olympiad in Informatics 2004)

Cho chuỗi X độ dài $n \leq 10^5$, tìm chuỗi Y ngắn nhất sao cho mọi ký tự trong X đều tồn tại một chuỗi con nào đó của X đúng bằng Y chứa vị trí ký tự đó. Hay nói cách khác, X là một phép ghép gối của một loạt các chuỗi Y

```
ababbababbababbabaababbaba (X)
ababbaba (Y)
  ababbaba
    ababbaba
      ababbaba
```

Thuật toán

Cách giải là sử dụng mảng hậu tố $SA(X) = (a_0, a_1, \dots, a_{n-1})$ kết hợp với một cấu trúc dữ liệu truy vấn phạm vi. Bắt đầu với $Y = X_0$. Những hậu tố có ký tự đầu bằng X_1 sẽ nằm trong một khoảng liên tiếp trong mảng hậu tố (từ vị trí L tới vị trí H). Vị trí ban đầu của các hậu tố này trong chuỗi sẽ được đánh dấu bởi số 1, những vị trí khác được đánh dấu bởi số 0.

Lần lượt thêm các ký tự X_1, X_2, \dots, X_{n-1} vào Y . Mỗi khi Y dài thêm một ký tự, sẽ có thêm những hậu tố của X không còn nhận Y làm tiền tố nữa, ta co ngắn phạm vi hoạt động $[L, H]$ lại và đánh dấu vị trí trong chuỗi của các hậu tố nằm ngoài phạm vi hoạt động bởi số 0. Thuật toán sẽ dừng ngay khi tới một bước mà độ dài chuỗi $Y \geq$ dãy nhiều số 0 liên tiếp nhất. Việc đo độ dài dãy gồm nhiều số 0 liên tiếp nhất có thể thực hiện trong thời gian $O(\log n)$ bằng một cấu trúc dữ liệu truy vấn phạm vi như segment trees. Toàn bộ thuật toán có độ phức tạp $O(n \log n)$.

3.6. Liên kết hậu tố (suffix links)

Thực ra trong cấu trúc của cây hậu tố, còn có một thành phần nữa gọi là các liên kết hậu tố (*suffix links*). Mỗi nút nhánh u của cây hậu tố chứa một con trỏ tới một nút v khác sao cho nếu $\bar{u} = \alpha S$ thì $\bar{v} = S$ (ở đây α là một ký tự còn S là một chuỗi)

Tất cả các thuật toán tuyến tính dựng cây hậu tố trực tiếp theo tôi biết đều phải sử dụng liên kết hậu tố. Tuy nhiên nếu ta dựng cây hậu tố từ mảng hậu tố và mảng tiền tố chung dài nhất, các liên kết hậu tố bị bỏ qua.

Các liên kết hậu tố đôi khi rất quan trọng trong một số thuật toán xử lý chuỗi. Vì vậy ta đặt vấn đề: cho cây hậu tố $ST(T)$ của một chuỗi T độ dài n , cần phải xây dựng toàn bộ các liên kết hậu tố.

Giải pháp:

Gán cho mỗi nút nhánh u của cây hậu tố một cặp (i, j) thỏa mãn: i, j là hai hậu tố ứng với hai lá nằm ở hai nhánh con khác nhau của u . Việc gán cặp (i, j) cho tất cả các nút nhánh có thể thực hiện trong thời gian $O(n)$ bằng thuật toán duyệt cây từ dưới lên: Để gán cặp hậu tố cho nút u , ta gán cặp hậu tố cho tất cả các con của u trước bằng đệ quy. Sau đó chọn u_1, u_2 là hai con bất kỳ của u , giả sử cặp hậu tố gán cho u_1 là (i_1, j_1) và cặp hậu tố gán cho u_2 là (i_2, j_2) , khi đó ta có thể lấy cặp (i_1, i_2) làm cặp lá ứng với u .

Với cặp hậu tố (i, j) của nút u , chúng ứng với hai lá nằm ở hai nhánh con khác nhau mà u là tiền bối chung thấp nhất của của hai lá đó, vậy nên tiền tố chung dài nhất của hai hậu tố i và j chính là \bar{u} mà ta ký hiệu là αS . Cũng từ đó, tiền tố chung dài nhất của hai hậu tố $i + 1$ và $j + 1$ phải là S . Xác định hai lá chứa hậu tố $i + 1$ và $j + 1$ và v là tiền bối chung thấp nhất của hai lá này. Ta có $\bar{v} = S$ tức là con trở liên kết từ u phải trở tới v .

Có rất nhiều thuật toán tìm tiền bối chung thấp nhất của hai nút trong thời gian $O(1)$, chẳng hạn các thuật toán đề xuất bởi Tarjan [12] hay Fisher [13]. Vì cây hậu tố có $O(n)$ nút, việc thiết lập toàn bộ các liên kết hậu tố có thể thực hiện trong thời gian $O(n)$. Bạn đọc có thể tham khảo thêm trong các tài liệu về hai bài toán LCA và RMQ và mối liên hệ giữa chúng.

3.7. Xâu con chung dài nhất

Bài toán tìm xâu con chung dài nhất (*longest common substring*) là một bài toán quan trọng trong xử lý xâu. Tên của bài toán đã nêu lên nội dung của nó: Cho hai xâu X, Y , cần tìm xâu Z độ dài lớn nhất vừa là xâu con của X , vừa là xâu con của Y .

Thuật toán 1

Giả sử $X = x_1x_2 \dots x_m$ và $Y = y_1y_2 \dots y_n$

Dựng cây hậu tố của Y . Bắt đầu từ gốc, ta duyệt các ký tự trong X và rẽ xuống nhánh con tương ứng trên cây hậu tố. Nếu tại một nút u nào đó không có nhánh con tương ứng để rẽ xuống, ta ghi nhận lại nút u cùng với độ sâu của nó (\bar{u} là xâu con dài nhất của X khớp với đoạn đầu xâu Y), tiếp theo ta nhảy theo liên kết hậu tố từ u sang nút v (\bar{v} xuất hiện trong Y tại vị trí y_2) và đi tiếp theo cách như vậy.

Sau khi duyệt hết xâu Y , nút có độ sâu lớn nhất ghi nhận được sẽ có nhãn là xâu con chung dài nhất cần tìm.

Độ phức tạp tính toán: $O(m + n)$.

Thuật toán 2

Bổ sung thêm ký tự cầm canh $\$$. Xét xâu $X@Y\$$, những hậu tố có vị trí đứng sau vị trí $@$ được gọi là hậu tố xanh (hậu tố của $Y\$$) và những hậu tố khác được gọi là hậu tố đỏ. Dựng cây hậu tố $X@Y\$$ trong đó các lá cũng được tô cùng màu với hậu tố tương ứng.

Bài toán trở thành tìm nút x sâu nhất mà nhánh cây gốc x chứa cả lá xanh và lá đỏ.

Độ phức tạp tính toán: $O(m + n)$. Có thể mở rộng để tìm LCS của nhiều xâu.

Thuật toán 3

Tô màu hậu tố tương tự như thuật toán 2. Dựng mảng hậu tố và mảng tiền tố chung dài nhất của $X@Y\$$. Trên mảng hậu tố, tìm vị trí i sao cho hậu tố a_i và hậu tố đứng liền trước (a_{i-1}) khác màu, chọn vị trí i có lcp_i lớn nhất.

Độ phức tạp tính toán: $O(m + n)$.

4. Kết luận

Cây hậu tố, mảng hậu tố và mảng tiền tố chung dài nhất là những cấu trúc dữ liệu có mối liên hệ chặt chẽ. Ngoài việc cung cấp nhiều phép toán quan trọng trong xử lý xâu, những kỹ thuật hay được áp dụng trong quá trình xây dựng cấu trúc dữ liệu cũng rất đáng chú ý.

Một trong những hướng nghiên cứu được quan tâm là sử dụng cây hậu tố để xử lý dữ liệu thuộc một bảng chữ cái lớn. Cấu trúc nút của cây có thể trở nên rất cồng kềnh nếu bảng chữ cái Σ lớn. Như ví dụ trong chuyên đề này, mỗi nút phải chứa một mảng các con trỏ liên kết tới các nút con. Kích thước của mảng con trỏ này đúng bằng $|\Sigma|$.

Khi kích thước bảng chữ cái lớn, mảng con trỏ có thể được thay thế bằng danh sách móc nối hay cây nhị phân tìm kiếm tự cân bằng để tiết kiệm bộ nhớ hơn, tuy nhiên điều đó có thể làm độ phức tạp tính toán của thuật toán bị phụ thuộc vào $|\Sigma|$. Bảng dưới đây tóm tắt về ảnh hưởng của cấu trúc nút lên thao tác rẽ nhánh (từ một nút đi sang nút con theo cạnh mang nhãn có ký tự đầu $\in \Sigma$)

Cấu trúc	Rẽ nhánh	Bộ nhớ
Mảng con trỏ	$O(1)$	$O(n \Sigma)$
Cây nhị phân tìm kiếm tự cân bằng	$O(\log \Sigma)$	$O(n)$
Danh sách móc nối	$O(\Sigma)$	$O(n)$
Danh sách động được sắp xếp	$O(\log \Sigma)$	$O(n)$

Khác với cây hậu tố, mảng hậu tố cũng như các thuật toán xây dựng mảng hậu tố lại không gặp khó khăn gì khi bảng chữ cái Σ lớn. Tuy vậy, có rất nhiều thao tác quan trọng trên cây hậu tố không thể dùng mảng hậu tố để thay thế được.

Năm 2004, nhóm nghiên cứu của Abouelhoda sau khi phân tích các ứng dụng đã có của cây hậu tố đã chỉ ra rằng: mọi kỹ thuật xử lý trên cây hậu tố trong các ứng dụng đã biết có thể quy về ba thao tác cơ bản [14]:

- Duyệt cây từ dưới lên, tổng hợp thông tin từ các nút con lên nút cha (bottom-up)
- Duyệt cây từ trên xuống, đi từ nút cha xuống nút con theo một nhãn cho trước (top-down)
- Từ một nút đi sang một nút khác theo liên kết hậu tố.

Từ đó, các tác giả đã đề xuất thêm những thuật toán xây dựng cấu trúc dữ liệu bổ sung. Những cấu trúc dữ liệu này sẽ kết hợp với mảng hậu tố để mô phỏng ba thao tác cơ bản trên. Cấu trúc dữ liệu mới này có tên là **mảng hậu tố tăng cường** (*enhanced suffix arrays*),

được chứng minh rằng có thể thay thế cho cây hậu tố trong tất cả các ứng dụng đã biết*. Kết quả này đã kéo theo nhiều nghiên cứu nhằm nâng cao tính hiệu quả của thuật toán xây dựng mảng hậu tố. Cho tới năm 2007, trong bài tổng quan của Puglisi trên tạp chí ACM Computing Survey [15], đã có tới 20 thuật toán xây dựng mảng hậu tố được đánh giá, chúng dựa trên những cách tiếp cận khác nhau, có những ưu/nhược điểm khác nhau tùy thuộc vào dạng dữ liệu.

Tuy nhiên, việc sử dụng mảng hậu tố để mô phỏng cây hậu tố đôi khi làm mất đi tính trực quan và gây khó khăn trong thiết kế thuật toán. Cũng trong nghiên cứu về mảng hậu tố tăng cường, các tác giả còn đề xuất một sửa đổi của cây hậu tố thành một cấu trúc dữ liệu mới gọi là **cây lcp-interval** (*lcp-interval trees*) [16] với đầy đủ các tính năng của cây hậu tố cũng như mảng hậu tố. Cấu trúc dữ liệu này khá dễ cài đặt, tiết kiệm bộ nhớ và quan trọng nhất là vẫn giữ nguyên được cấu trúc cây. Năm 2008, nhóm nghiên cứu của Kim [16] còn tìm ra nhiều tính chất quan trọng của cây lcp-interval so với cây hậu tố truyền thống, đưa ra phương pháp cài đặt trong trường hợp bảng chữ cái lớn và đặc biệt là đưa ra được thuật toán tính các liên kết hậu tố một cách đơn giản và trực tiếp, không cần thông qua truy vấn LCA hay RMQ.

Trong các kỳ thi lập trình có sự hạn chế thời gian, ngoài tiêu chí về tính hiệu quả trong việc lựa chọn thuật toán, luôn phải quan tâm tới tính đơn giản. Những cấu trúc dữ liệu trong chuyên đề này không dễ cài đặt, chính vì vậy thí sinh không nên lạm dụng chúng trong phòng thi. Ví dụ: thay vì mất ~100 dòng lệnh để cài đặt mảng hậu tố (hoặc hơn với cây hậu tố) chỉ để giải quyết bài toán xác định xâu con, thí sinh có thể cài đặt thuật toán KMP với chưa tới 20 dòng lệnh nhưng lại đạt hiệu quả cao hơn rất nhiều.

Nếu thí sinh đã biết về cây/mảng hậu tố và gặp một bài toán có thể giải quyết triệt để bằng các cấu trúc dữ liệu này, thí sinh ít nhiều sẽ có lợi thế về tâm lý bởi việc còn lại chỉ là vấn đề thời gian. Tuy nhiên lợi thế này sẽ nhanh chóng tạo ra bất lợi nếu:

- Ràng buộc dữ liệu và yêu cầu của bài toán cho phép thiết kế thuật toán đơn giản và hiệu quả hơn nhiều. Nếu không phân tích kỹ đề bài, thí sinh sẽ mất cơ hội tìm ra thuật toán tốt và bị thiệt về thời gian. Nói chung nếu đọc đề ẩu hoặc không phân tích kỹ đề bài thì càng biết nhiều sẽ càng bất lợi.
- Khả năng cài đặt của thí sinh không tốt, quy trình kiểm thử của thí sinh không cẩn thận, hoặc thí sinh không đủ thời gian để kiểm thử. Cấu trúc dữ liệu hiệu quả nhưng cài đặt sai có thể mất nhiều điểm hơn cả những chương trình cài đặt thuật toán tầm thường.

Để sử dụng các thuật toán hay cấu trúc dữ liệu phức tạp một cách hiệu quả (nói riêng với cây/mảng hậu tố), thí sinh phải phân tích kỹ ràng buộc và yêu cầu bài toán và cố gắng tìm thuật toán đơn giản. Trong trường hợp bắt buộc phải sử dụng giải pháp phức tạp, cần lường trước thời gian lập trình, gỡ rối và kiểm thử. Ngoài ra, việc cài đặt thuật toán tầm

* Nguyên văn "every algorithm that uses a suffix tree as data structure can **systematically be replaced** with an algorithm that uses an enhanced suffix array and solves the same problem in the same time complexity".

thường cũng là cần thiết để đối sánh kết quả và tránh mất điểm quá nhiều nếu không kịp thời gian hoàn thiện.

Tài liệu tham khảo

- [1] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490-499, 1960.
- [2] D. R. Morrison, "PATRICIA - Practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM*, vol. 15, no. 4, pp. 514-534, 1968.
- [3] P. Weiner, "Linear pattern matching algorithms," in *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, Washington, DC, USA, 1973.
- [4] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, no. 2, pp. 262-272, 4 1976.
- [5] A. Slissenko, "Detection of periodicities and string-matching in real time," *Journal of Soviet Mathematics*, vol. 22, no. 3, pp. 1316-1386, 1983.
- [6] S. R. Kosaraju, "Real-time pattern matching and quasi-real-time construction of suffix trees," in *Proceedings of the 26th annual ACM symposium on theory of computing*, 1994.
- [7] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249-260, 1995.
- [8] U. Manber and Myers, Gene, "Suffix arrays: a new method for on-line string searches," in *Proceedings of the first annual ACM-SIAM symposium on discrete algorithms*, San Francisco, 1990.
- [9] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," in *Proceedings of the 13th international conference on automata, languages and programming*, 2003.
- [10] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," *Proceedings of the 14th annual symposium on combinatorial pattern matching*, pp. 200-210, 2003.
- [11] T. Kasai, G. Lee, H. Arimura, S. Arikawa and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, London, 2001.
- [12] D. Harel and R. E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 338-355, 1984.
- [13] J. Fische and V. Heun, "Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE," *Lecture notes in Computer Science*, vol. 4009, pp. 36-48, 2006.

- [14] M. I. Abouelhoda, Kurtz, Stefan and Ohlebusch, Enno, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53-86, 2004.
- [15] S. J. Puglisi, W. F. Smyth and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Survey*, vol. 39, no. 2, pp. 1-31, 2007.
- [16] D. K. Kim, M. Kim and H. Park, "Linearized Suffix Tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays," *Algorithmica*, vol. 52, no. 3, pp. 350-377, 2008.
- [17] Gusfield, Dan, *Algorithms on strings, trees, and sequences: computer science and computational biology*, New York, USA: Cambridge University Press, 1997.
- [18] R. Giegerich, Kurtz, Stefan and Stoye, Jens, "Efficient implementation of lazy suffix trees," in *Proceedings of the 3rd International Workshop on Algorithm Engineering*, London, UK, 1999.