CÁC KỸ THUẬT CƠ BẢN ĐỂ TĂNG TỐC CHƯƠNG TRÌNH

Việc lưu trữ thông tin cần nhớ sao cho có thể lấy lại chúng một cách nhanh nhất là một trong các kỹ năng cơ bản đầu tiên để có thể có một chương trình hiệu quả. Tuy vậy việc nhớ cái gì và lưu trữ như thế nào lại đòi hỏi sự nhạy bén toán học của học sinh. Điều này lại chỉ được hình thành sau khi học sinh tiếp xúc với một hệ thống các bài toán được tổ chức cẩn thận. Hệ thống này giúp học sinh xây dựng được các thói quen tư duy cơ bản cũng như các kỹ thuật cơ bản trong lập trình. Dưới đây, tôi trình bày một hệ thống các bài tập được phân loại kỹ lưỡng qua nhiều năm giảng dạy nhằm mục đích hình thành cho các em các kỹ năng nói trên.

1. Kỹ thuật nhớ

Đây là kỹ thuật đơn giản, các thông tin cần nhớ được lưu trữ vào một mảng ở vị trí thích hợp. Khi cần, chỉ việc lấy thông tin đó ra trong khoảng thời gian O(1).

Bài toán 1: Cho mảng n số nguyên dương a_1, a_2, ..., a_n ($n \le 10^6$, $a_i \le 10^6$) và số nguyên dương S ($S \le 10^6$). Hãy đếm xem có bao nhiêu cặp (a_i,a_j) thỏa mãn $a_i+a_j=S$.
Thuật toán $O(n^2)$ đơn giản đề giải bài toán này là:

```
ds:=0;
for j:=1 to n do
for i:=1 to j-1 do
if a[i]+a[j]=S then ds:=ds+1;
writeln(ds);
```

Để cải tiến, chúng ta xem xét đẳng thức điều kiện a[i]+a[j]=S. Nếu j cố định thì a[i]=S-a[j] cũng cố định. Và như vậy có thể thấy rằng vòng lặp bên trong chẳng qua chỉ là *đếm xem có bao nhiêu phần tử bằng S-a[j]*. Vậy nếu ta nhớ được số lượng này thì xem như không cần vòng lặp bên trong để đếm nữa!. Để ý đến điều kiện đầu bài $1 \le a[i] \le 10^6$ ta hoàn toàn có thể lập mảng nhớ:

```
var c: array[1..1000000] of longint;
```

với c[x] là số lượng các phần tử bằng x đã xuất hiện. Ta có chương trình hiệu quả để giải quyết như sau:

```
fillchar(c,sizeof(c),0);
for j:=1 to n do
begin
  if (S-a[j]>0) then inc(ds,c[S-a[j]]);
  inc(c[a[j]]);
end;
```

Thuật toán trên có độ phức tạp O(n) và đáp ứng dược yêu cầu bài toán đề ra về mặt thời gian. Tuy nhiên có thể thấy chi phí bộ nhớ tăng lên. Có thể nhận xét điều này để học sinh thấy rằng *muốn có một chương trình chạy nhanh hơn thì chi phí bộ nhớ phải tốn hơn*.

Bài toán 2: Cho dãy số $a_1, a_2, ..., a_n$ ($1 \le n \le 10^6$, $|a_i| \le 10^9$). Hãy tìm dãy con của dãy số đã cho gồm các phần tử liên tiếp sao cho tổng các phần tử của dãy con này là lớn nhất.

Đây là một trong những bài toán điển hình cho việc cải tiến chương trình sao cho mức độ hiệu quả ngày càng cao hơn.

Ta xét thuật toán đơn giản nhất để giải bài toán trên có độ phức tạp $O(n^2)$:

```
ds:=0;
for j:=1 to n do
  for i:=1 to j do
    begin
      T:=0;
      for k:=I to j do T:=T+a[k];
      if T>ds then ds:=T;
    end;
```

Thuật toán trên hoàn toàn tự nhiên. Tuy nhiên với độ phức tạp $O(n^3)$ thì nó chỉ cho kết quả trong thời gian 1 giây khi $n \le 100$. Cần phải có các thuật toán tinh tế hơn.

Nhận xét rằng để tính tổng T=a[i]+a[i+1]+…a[j] ta có thể viết

T=(a[1]+a[2]+…+a[j])-(a[1]+a[2]+…+a[i-1]) = s[j]-s[i-1]

trong đó s[k]=a[1]+…a[k]

Bằng cách chuẩn bị trước mảng S trong O(n) ta có thể thay thế vòng lặp trong cùng của thuật toán trên bằng 1 lệnh. Ta có chương trình cải tiến sau:

```
ds:=a[1];
s[0]:=0;
for i:=1 to n do s[i]:=s[i-1]+a[i];
for j:=1 to n do
  for i:=1 to j do
    if s[j]-s[i-1]>ds then ds:=s[j]-s[i-1];
```

Độ phức tạp của thuật toán thứ hai là $O(n^2)$ và dùng thuật toán này có thể giải quyết bài toán với $n \le 2000$.

Để tăng tốc chương trình hơn nữa, đến đây chúng ta lại sử dụng kỹ thuật nhớ như trong bài toán 1. Xét vòng lặp trong cùng (i). Nếu j cố định thì vòng lặp này chẳng qua tìm chỉ số I để s[j]-s[i-1] đạt giá trị lớn nhất. Điều này tương đương với việc tìm giá trị nhỏ nhất của s[i-1] với i=1,2,…,j hay là tìm giá tị nhỏ nhất của s[0],s[1],…,s[j-1]. Bằng cách dùng thêm một biến min để lưu giá trị nhỏ nhất chúng ta có được một thuật toán O(n) đáp ứng yêu cầu của bài toán như sau:

```
s[0]:=0;
for i:=1 to n do s[i]:=s[i-1]+a[i];
ds:=a[1];
min:=0;
```

```
for j:=1 to n do
  begin
    if s[j]-min>ds then ds:=s[j]-min;
    if s[j]<min then min:=s[j];
  end;
```

Có rất nhiều điều có thê tổng kết và mở rộng sau bài này. Trước tiên là củng cố lại kỹ thuật nhớ cho học sinh. Ngoài ra, để tính tổng các số trong đoạn [i,j] ta luôn đưa về hiệu giữa tổng các số trong đoạn [1,j] và tổng các số trong đoạn [1,i-1]. Kỹ thuật này cũng thường được dùng trên mảng hai chiều: Để tính tổng các số trong một hình chữ nhật với đỉnh trên bên trái (i1,j1) còn đỉnh dưới bên phải (i2,j2) ta lập mảng s[u,v] bằng tổng các số trong hình chữ nhật (1,1,u,v). Khi đó tổng các số trong hình chữ nhật (i1,j1,i2,j2) là:

T(i1,j1,i2,j2)=s[i2,j2]-s[i1-1,j2]-s[i2,j1-1]+s[i1-1,j1-1]

với s[u,v] có thể tính theo công thức:

s[u,v]=s[u-1,v]+s[u,v-1]-s[u-1,v-1]+a[u,v]

Bài toán 3: (mở rộng của bài toán 2) Cho hình chữ nhật m hàng, n cột trong đó tại các vị trí giao của hàng i cột j chứa số a[i,j]. Hãy tìm hình chữ nhật con của hình chữ nhật đã cho có các cạnh song song với các cạnh của hình chữ nhật ban đầu và có tổng lớn nhất.

Sau khi đã hướng dẫn học sinh làm bài toán 2 thì việc đưa ra thuật toán hiệu quả để giải bài toán này không phải là công việc khó khăn. Chỉ cần gợi ý là sử dụng phương pháp của bài toán 2, các em học sinh nhanh chóng nhận ra rằng khi hàng trên và hàng dưới cùng của hình chữ nhật cố định đây chính là bài toán 2 và các em đã cho ngay được một phương án hiệu quả:

```
ds:=a[1,1];
for i1:=1 to m do
  for i2:=i1 to m do
    begin
      b[0]:=0;
      for k:=1 to n do b[k]:=b[k-1]+T(i1,k,i2,k);
      min:=0;
      for k:=1 to n do
        begin
          if b[k]-min>ds then ds:=b[k]-min
          if min>b[k] then min:=b[k];
        end;
    end;
```

Độ phức tạp thuật toán trên là $O(m^2n)$

Một điều thú vị là kỹ thuật cố định hàng trên và hàng dưới là kỹ thuật phổ biến để đưa việc giải bài toán hai chiều thành bài toán một chiều

Bài toán 4: Cho dãy nhị phân độ dài n. Hãy đếm xem có bao nhiêu dãy con của dãy đã cho có số lượng số 1 bằng số lượng số 0

Với việc làm quen kỹ thuật tính tổng trên đoạn [i,j] có thể nhanh chóng đưa ra một thuật toán $O(n^2)$ để giải bài toán trên như sau:

```
dem:=0;
for j:=1 to n do
  for i:=1 to j do
    if s1[j]-s1[i-1]=s0[j]-s0[i-1] then inc(dem);
```

Ở đây s1[k], s0[k] lần lượt là số lượng số 1, số lượng số 0 trong đoạn [1,k]. Hai mảng này có thể chuẩn bị trước trong thời gian O(n).

Để có thể cải tiến chương trình trên, ở đây sử dụng kỹ thuật hay dùng trong toán học là phân li các ẩn số:
Nhận xét rằng biểu thức s1[j]-s1[i-1]=s0[j]-s0[i-1] tương đương với s1[j]-s0[j]=s1[i-1]-s0[i-1] (chuyển các số hạng có j về một vế, số hạng có i về về còn lại). Ta đi đến kết luận quan trọng rằng khi j cố định, biến dem sẽ tăng một lượng đúng bằng số lượng các vị trí i trước đó có s1[i]-s0[i]=s1[j]-s0[j]. Sử dụng kỹ thuật mảng nhớ như trong bài toán 1 chúng ta có ngay thuật toán O(n):

```
dem:=0;
c[0]:=1;
for j:=1 to n do
  begin
    inc(dem,c[s1[j]-s0[j]]);
    inc(c[s1[j]-s0[j]]);
  end;
```

2. Kỹ thuật duy trì mảng sắp xếp

Trong kỹ thuật nhớ, mỗi lần lấy thông tin nhớ để xử lý chúng ta chỉ ra đích xác phần tử cần lấy thông qua chỉ số của nó. Tuy vậy, trong một số trường hợp khác phần tử nhớ cần lấy ra không xuất hiện tường minh (vì không đủ bộ nhớ chứa nó) mà thường nằm trong một dải giá trị nào đó. Nếu như duy trì được mảng nhớ ở dạng sắp xếp tăng hoặc giảm dần thì kỹ thuật tăng tốc hay gặp ở đây là kỹ thuật tìm kiếm nhị phân hoặc đơn giản là dò tuyến tính trên mảng nhớ.

Bài toán 5: Cho dãy $a_1, a_2, \ldots, a_n$. Hãy đếm xem có bao nhiêu dãy con của dãy đã cho gồm các số hạng liên tiếp mà có tổng bằng 0.
Bằng cách thử tất cả các dãy con có thể có ta có được thuật toán $O(n^2)$ như sau:

```
dem:=0;
for j:=1 to n do
  for i:=1 to j do
    if s[j]=s[i-1] then inc(dem);
```

Tất nhiên, nếu giá trị các a[i] nhỏ kéo theo giá trị các s[i] cũng nhỏ thì kỹ thuật nhớ ở phần trên có thể áp dụng. Tuy nhiên, khi a[i] lớn thì không thể làm được như vậy bởi vì không đủ bộ nhớ (!!!). Nhận xét rằng các thông tin cần nhớ ở đây là mảng s và nếu mảng này được sắp

xếp tăng dần thì bài toán đơn giản chỉ là đếm xem có bao nhiêu cặp (s[i], s[j]) bằng nhau. Sau khi sắp xếp lại mảng S. Việc đếm có thể tính trong O(n) như sau:

```
u:=1;
for v:=2 to n+1 do
  if s[v]>s[u] then
    begin
      k:=v-u;
      inc(dem,(k-1)*k div 2);
      u:=v;
    end;
```

Ở đây s[n+1]=vc là phần tử đủ lớn để cầm canh.

Độ phức tạp thuật toán tổng cộng cho các phần là O(nlogn) chủ yếu phụ thuộc vào thuật toán sắp xếp được lựa chọn.

Bài toán 6: Cho dãy $a_1, a_2, \ldots, a_n$. Hãy tìm dãy con tăng dài nhất (các phần tử của dãy con tăng không nhất thiết là liên tiếp) của dãy đã cho.
Nếu gọi f[i] là độ dài của dãy con tăng dài nhất kết thúc tại a[i] ta có công thức truy hồi như sau:

$$\begin{cases} f[0] = 1 \\ f[i] = \max\{f[k] : k < i, a[k] < a[i]\} + 1 \end{cases}$$

Ở đây a[0]=-vc là một số đủ nhỏ để cầm canh.
Nếu như biết f[i] thì đáp số của bài toán sẽ là max{f[i]: i=1,2,…,n}.
Để tính f[i], ,một thuật toán $O(n^2)$ thường dùng như sau:

```
f[0]:=0;
for i:=1 to n do
  begin
    f[i]:=1;
    for k:=i-1 downto 0 do if (a[k]<a[i]) then
    if f[i]<f[k]+1 then f[i]:=f[k]+1;
  end;
```

Kỹ thuật nhớ trình bày ở phần 1 không áp dụng được ở đây, đơn giản vì chúng ta cần chọn ra một giá trị nhớ thuộc một "miền" nào đó.

Nhận xét rằng có thể tồn tại nhiều giá trị k để cuối cùng f[i]=f[k]+1. Nếu như bằng cách nào đó chỉ lưu trứ 1 trong số chúng thì tốc độ thuật toán sẽ được cải thiện vì khi đó thay vì duyệt qua hết tất cả các giá trị f đã có trước đó, chúng ta chỉ cần duyệt qua các giá trị cần nhớ thôi.

Vì chúng ta chỉ quan tâm đến các dãy con tăng nên hiển nhiên là nếu với hai dãy con tăng độ dài cùng bằng u ta chỉ cần nhớ dãy con tăng có phần tử cuối nhỏ hơn (vì nếu nhớ thêm chắc chắn là thừa). Chính vì vậy, thay vì nhớ mảng f ta nhớ bằng bảng h: array[1..n] of longint; trong đó h[i] là giá trị của phần tử cuối cùng trong dãy con tăng độ dài i đã tìm được.

Nếu quan sát tỉ mỷ, chúng ta phát hiện rằng h[1]≤h[2]≤…≤h[n] vì một dãy con tăng độ dài u hiển nhiên là dãy con tăng độ dài u-1 và như vậy việc tìm giá trị nhớ thích hợp trên mảng h có thể thực hiện bằng tìm kiếm nhị phân:

```
for i:=1 to n do h[i]:=vc;
h[0]:=-vc;
f[0]:=0;
for i:=1 to n do
  begin
    u:=first(a[i]);
    f[i]:=u;
    h[u]:=a[i];
  end;
```

Thuật toán thực hiện trong thời gian O(nlong)

Bài toán 7: Cho hai dãy $a_1, a_2, \ldots, a_m$ và $b_1, b_2, \ldots, b_n$. Hãy tìm giá trị nhỏ nhất của tổng $|a_i + b_j|$.

Đây là bài toán mà thuật toán tự nhiên là khá đơn giản:

```
for i:=1 to m do
  for j:=1 to n do
    if abs(a[i]+b[j])<ds then ds:=abs(a[i]+b[j]);
```

Để có được thuật toán hiệu quả hơn, nhận xét rằng vòng lặp trong (j) chẳng qua là tìm trong mảng B phần tử "gần" với -a[i] nhất. Như vậy nếu mảng B được sắp xếp tăng thì thay vì tìm kiếm tuyến tính ta có thể sử dụng kỹ thuật tìm kiếm nhị phân:

```
for i:=1 to n do
  begin
     u:=first(-a[i]);
     if b[u]-b[u-1]<ds then ds:=b[u]-b[u-1];
     if b[u+1]-b[u]<ds then ds:=b[u+1]-b[u];
  end;
```

Ở đây b[0] gán cho một giá trị đủ nhỏ còn b[n] gàn cho giá trị đủ lớn như là một kỹ thuật cầm canh.

Độ phức tạp thuật toán là O(nlogn)

Kỹ thuật tìm kiếm nhị phần trên mảng nhớ là một trong những kỹ thuật cơ bản. Tuy vậy, trong trường hợp tập hợp nhớ luôn biến động luôn phá vỡ tính được sắp thì kỹ thuật này không hiệu quả. Trong những trường hợp như vậy, người ta thường sử dụng các kỹ thuật lưu trữ dữ liệu cao cấp hơn như cây tìm kiếm nhị phân (BST) hoặc mảng băm (hash). Tuy nhiên tử tưởng của tìm kiếm nhị phân vẫn được dùng trong các tình huống này (mở rộng của các hàm first, last trên BST…)

3. Kỹ thuật sử dụng hàng đợi

Trong các tình huống các phần tử của mảng nhớ luôn biến động nhưng lại tuân theo qui luật là chỉ thêm/bớt vào ở các vị trí đầu tiên/cuối cùng thì việc sử dụng một hàng đợi hai

đầu là sự lựa chọn phù hợp. Khó khăn chính ở đây là làm thế nào có thể mô tả được hàng đợi hai đầu đó. Không có một nguyên tắc chung nào trong trường hợp này cả. Tuy nhiên, có thể hệ thống một số tình huống hay gặp thông qua các bài toán sau:

Bài toán 8: Cho dãy n số nguyên a_1, a_2, ..., a_n và m câu hỏi truy vấn trên dãy này. Câu hỏi thứ i có dạng (u_i,v_i) với ý nghĩa là hỏi xem giá trị nhỏ nhất trong các số $a[u_i]$, $a[u_i+1]$,...,$a[v_i]$ bằng bao nhiêu. Biết rằng:

$u_1 \le u_2 \le ... \le u_m$

$v_1 \le v_2 \le ... \le v_m$.

Nếu không xét đến hai điều kiện cuối cùng trong đề bài thì thuật toán O(mn) là hiển nhiên:

```
for i:=1 to m do
  begin
    min:=a[u[1]];
    for j:=u[i]+1 to v[i] do
    if min>a[j] then min:=a[j];
    writeln(min);
  end;
```

Tuy nhiên điều kiện ràng buộc của u[i], v[i] cho phép chúng ta nghĩ đến việc tận dụng các thông tin có được trong quá trình tìm min của câu hỏi trước để xử lý tìm min cho câu hỏi sau (lưu ý rằng chiến lược kiểu như thế này cũng là một trong các chiến lược thường được áp dụng khi xây dựng các thuật toán hiệu quả từ các thuật toán tầm thường).

Giả sử ta đã xét xong truy vấn i-1 và bây giờ xét truy vấn i. Khi từ truy vấn i-1 sang truy vấn i chúng ta đã xét thêm các phần tử a[v[i-1]+1],...,a[v[i]] và loại đi các phần tử a[u[i-1]+1],...,a[u[i]].

Mỗi khi bỏ đi một phần tử, nếu giá trị của nó lớn hơn giá trị min thì giá trị min mới không thay đổi. Tuy nhiên nếu giá trị của phần tử bỏ đi bằng giá trị min thì hiển nhiên giá trị min mới sẽ nhận giá trị nhỏ tiếp theo (giá trị này cũng có thể bằng giá trị min cũ - tất nhiên). Chính vì vậy bên cạnh việc theo dõi giá trị nhỏ nhất chúng ta cần theo dõi thêm giá trị nhỏ thứ nhì, thứ ba, ... Tức là xây dựng một dãy các phần tử nhớ có giá trị tăng dần. Dưới đây ta sẽ chỉ ra dãy các phần tử nhớ này hoạt động như một hàng đợi kép:

+ Nếu phần tử bó đi có giá trị bằng giá trị nhỏ nhất (vị trí front) thì hiển nhiên giá trị nhỏ nhất sẽ bằng giá trị tiếp theo. Điều này tương đương với việc lấy ra phần tử ở đầu hàng đợi.

+Nếu phần tử thêm vào có giá trị lớn hơn phần tử cuối cùng (lớn nhất) thì nó là giá trị lớn tiếp theo, do vậy nó được lưu vào vị trí cuối cùng trong hàng đợi. Trong trường hợp ngược lại nhận xét rằng các giá trị đã có trong hàng đợi lớn hơn giá trị mới sẽ không bao giờ là giá trị nhỏ nhất của một truy vấn nào. Chính điều này cho phép chúng ta cho ra khỏi hàng đợi các giá trị này. Hiển nhiên các giá trị lây ra luôn ở cuối hàng đợi.

Như vậy ta có một hàng đợi hai đầu quản lý các giá trị nhỏ nhất theo thứ tự tăng dần. Vì mỗi phần tử được đưa vào hàng đợi 1 lần và lấy ra 1 lần nên tổng thời gian thực hiện của toàn bộ thuật toán là O(n):

```
front:=1; back:=0;
u[0]:=0; v[0]:=0;
for i:=1 to m do
  begin
    for k:=v[i-1]+1 to v[i] do
      begin
        while (front<=back) and (a[k]>q[back]) do dec(back);
        inc(back); q[back]:=a[k];
      end;
      for k:=u[i-1]+1 to u[i] do
        if a[k]=q[front] then inc(front);
      writeln(q[front]);
  end;
```

Mặc dù phát biểu của bài toán 8 không tự nhiên (do điều kiện ràng buộc) tuy nhiên theo kinh nghiệm của tôi đây là một dạng bài cơ bản bởi vì tất cả các tình huống có sử dụng hàng đợi hai đầu đều đưa về dạng trên (nếu tìm max thì lưu hàng đợi các giá trị giảm dần, nếu tìm min thì lưu hàng đợi các giá trị tăng dần)

Bài toán 9: Cho dãy số $a_1, a_2, ..., a_n$. Hãy xây dựng hai mảng prev, next với ý nghĩa prev[i] = chỉ số của phần tử "xa" i nhất về phía đầu dãy có giá trị lớn hơn hoặc bằng a[i] và next[i] = chỉ số của phần tử "xa" i nhất về phía cuối dãy có giá trị lớn hơn hoặc bằng a[i].

Đây không phải là bài toán thực sự khi kiểm tra. Tuy nhiên nó lại là bài toán "nền" mà việc giải cũng như cách lập luận đến lời giải của nó là cơ sở để giải quyết rất nhiều bài toán khác.

Ta chỉ xét việc tìm prev. Việc tìm next được thực hiện một cách tương tự.
+Nếu a[i]>a[i-1] thì tất nhiên prev[i]=i;
+Nếu a[i]≤a[i-1] thì do nhận xét rằng với u<i mà a[u]≥a[i] thì
prev[i]≤prev[u]≤u<i
Do vậy prev[i]≤prev[i-1]. Ngoài ra với mọi j>i thì pre[i-1] không bao giờ bằng prev[j] vậy ta có thể bỏ giá trị này đi không nhớ nữa.

Điều này làm cho chúng ta nghĩ đến việc lưu các giá trị prev như một hàng đợi. Điều lý thú là hàng đợi này chỉ thêm hoặc bớt phần tử ở phía cuối (một hàng đợi như vậy thường được gọi là một ngăn xếp:

```
back:=0;
for i:=1 to n do
  begin
    if a[i]>a[i-1] then
      begin
        prev[i]:=i;
```

```
        inc(back); q[back]:=i;
      end
    else
      begin
        while a[q[back]]>=a[i] do dec(back);
        inc(back);
        prev[i]:=prev[q[back]];
        q[back]:=i;
      end;
  end;
```

Do mỗi chỉ số chỉ đưa vào và lấy ra khỏi ngăn xếp không quá 1 lần nên độ phức tạp của thuật toán trên là O(n).

Bài toán 10: Cho mảng hai chiều m hàng và n cột trong đó các phần tử chỉ nhận giá trị 0 hoặc 1. Hãy tìm hình chữ nhật chứa cùng 1 số (0 hoặc 1) sao cho diện tích là lớn nhất.

Ta chỉ cần xét các hình chữ nhật chứa toàn số 1, đối với các hình chữ nhật chứa toàn số 0 tình hình hoàn toàn tương tự.

Dễ dàng có được thuật toán $O(m^2n^2)$ bằng cách thử tất cả các hình chữ nhật có thể. Nhận xét rằng nếu cố định cạnh dưới của hình chữ nhật là hàng i thì chiều cao của hình chữ nhật có diện tích lớn nhất sẽ bằng giá trị của số lượng số 1 liên tiếp từ hàng i hất lên trên của một cột nào đó (vì nếu không ta luôn có thể mở rộng được hình chữ nhật lớn hơn) và bằng cách sử dụng kết quả bài 9 ta có được thuật toán O(mn) như sau:

Với mỗi hàng i xây dựng mảng h[1], ..., h[n] với h[j]=số lượng số 1 liên tiếp bắt đầu từ hàng i hất lên trên (mảng này còn gọi là mảng độ cao). Ta thử hình chữ nhật với chiều cao là một giá trị h[j] nào đó. Hiển nhiên mép trái của hình chữ nhật là prev[i] còn mép phải là next[i] (hai mảng này được xây dựng cùng thời gian với h và diện tích của hình chữ nhật là h[j](next[j]-prev[j]+1):

```
for i:=1 to m do
  begin
    for j:=1 to n do if a[i,j]=0 then h[j]:=0 else h[j]:=h[j]+1;
    // xay dung mang prev, next
    for j:=1 to n do
      begin
        t:=h[j]*(next[j]-prev[j]+1);
        if t>ds then ds:=t;
      end;
  end;
```

Các bài tập sử dụng hàng đợi kép (double queue) rất đa dạng và phong phú. Tuy nhiên có thể thấy rằng phần lớn trong số đó là sử dụng các kỹ thuật đã nêu

*

* *

Hệ thống 10 bài toán nhằm cho ra thuật toán hiệu quả đã được tôi thử nghiệm trong quá trình giảng dạy các chuyên đề lập trình cho học sinh. Một trong các kết quả ấn tượng là hầu hết các em, sau khi hoàn thành học tập chuyên đề này đều có phản ứng rất tích cực trước các bài toán thuộc nhóm tương tự. Một số em đã có được nhiều kết quả độc đáo.

Các kỹ thuật trình bày ở trên mới chỉ là các kỹ thuật mở đầu trong việc xây dựng các cấu trúc dữ liệu thích hợp phục vụ các bài toán khác nhau. Chẳng hạn nếu như tập hợp các biến nhớ liên tục thay đổi và không duy trì được như một mảng được sắp xếp hoặc như một hàng đợi kép thì lúc đó những cấu trúc phức tạp hơn như đống (heap), cây tìm kiếm nhị phân (BST) hoặc mảng băm (hash) sẽ dược sử dụng. Các kỹ thuật này là cao cấp và thực tế là quá sức tự nghiên cứu đối với các em học sinh.