

## §8. SẮP XẾP (SORTING)

### 8.1. BÀI TOÁN SẮP XẾP

Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự nhất định. Chẳng hạn như thứ tự tăng dần (hay giảm dần) đối với một dãy số, thứ tự từ điển đối với các từ v.v... Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng Tin học với các mục đích khác nhau: sắp xếp dữ liệu trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu v.v...

Nói chung, dữ liệu có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước: Một tập các đối tượng cần sắp xếp là tập các bản ghi (records), mỗi bản ghi bao gồm một số trường (fields) khác nhau. Nhưng không phải toàn bộ các trường dữ liệu trong bản ghi đều được xem xét đến trong quá trình sắp xếp mà chỉ là một trường nào đó (hay một vài trường nào đó) được chú ý tới thôi. Trường như vậy ta gọi là **khoá (key)**. Sắp xếp sẽ được tiến hành dựa vào giá trị của khoá này.

Ví dụ: Hồ sơ tuyển sinh của một trường Đại học là một danh sách thí sinh, mỗi thí sinh có tên, số báo danh, điểm thi. Khi muốn liệt kê danh sách những thí sinh trúng tuyển tức là phải sắp xếp các thí sinh theo thứ tự từ điểm cao nhất tới điểm thấp nhất. Ở đây khoá sắp xếp chính là điểm thi.

| STT | SBD  | Họ và tên    | Điểm thi |
|-----|------|--------------|----------|
| 1   | A100 | Nguyễn Văn A | 20       |
| 2   | B200 | Trần Thị B   | 25       |
| 3   | X150 | Phạm Văn C   | 18       |
| 4   | G180 | Đỗ Thị D     | 21       |

Khi sắp xếp, các bản ghi trong bảng sẽ được đặt lại vào các vị trí sao cho giá trị khoá tương ứng với chúng có đúng thứ tự đã định. Vì kích thước của toàn bản ghi có thể rất lớn, nên nếu việc sắp xếp thực hiện trực tiếp trên các bản ghi sẽ đòi hỏi sự chuyển đổi vị trí của các bản ghi, kéo theo việc thường xuyên phải di chuyển, copy những vùng nhớ lớn, gây ra những tổn phí thời gian khá nhiều. Thường người ta khắc phục tình trạng này bằng cách xây dựng một bảng khoá: Mỗi bản ghi trong bảng ban đầu sẽ tương ứng với một bản ghi trong **bảng khoá**. Bảng khoá cũng gồm các bản ghi nhưng mỗi bản ghi chỉ gồm có hai trường:

Trường thứ nhất chứa khoá

Trường thứ hai chứa liên kết tới một bản ghi trong bảng ban đầu, tức là chứa một thông tin đủ để biết bản ghi tương ứng với nó trong bảng ban đầu là bản ghi nào.

Sau đó, việc sắp xếp được thực hiện trực tiếp trên bảng khoá, trong quá trình sắp xếp, **bảng chính không hề bị ảnh hưởng gì**, việc truy cập vào một bản ghi nào đó của bảng chính vẫn

có thể thực hiện được bằng cách dựa vào trường liên kết của bản ghi tương ứng thuộc bảng khoá.

Như ở ví dụ trên, ta có thể xây dựng bảng khoá gồm 2 trường, trường khoá chứa điểm và trường liên kết chứa số thứ tự của người có điểm tương ứng trong bảng ban đầu:

| Điểm thi | STT |
|----------|-----|
| 20       | 1   |
| 25       | 2   |
| 18       | 3   |
| 21       | 4   |

Sau khi sắp xếp theo trật tự điểm cao nhất tới điểm thấp nhất, bảng khoá sẽ trở thành:

| Điểm thi | STT |
|----------|-----|
| 25       | 2   |
| 21       | 4   |
| 20       | 1   |
| 18       | 3   |

Dựa vào bảng khoá, ta có thể biết được rằng người có điểm cao nhất là người mang số thứ tự 2, tiếp theo là người mang số thứ tự 4, tiếp nữa là người mang số thứ tự 1, và cuối cùng là người mang số thứ tự 3, còn muốn liệt kê danh sách đầy đủ thì ta chỉ việc đối chiếu với bảng ban đầu và liệt kê theo thứ tự 2, 4, 1, 3.

Có thể còn cải tiến tốt hơn dựa vào nhận xét sau: Trong bảng khoá, nội dung của trường khoá hoàn toàn có thể suy ra được từ trường liên kết bằng cách: Dựa vào trường liên kết, tìm tới bản ghi tương ứng trong bảng chính rồi truy xuất trường khoá trong bảng chính. Như ví dụ trên thì người mang số thứ tự 1 chắc chắn sẽ phải có điểm thi là 20, còn người mang số thứ tự 3 thì chắc chắn phải có điểm thi là 18. Vậy thì bảng khoá có thể loại bỏ đi trường khoá mà chỉ giữ lại trường liên kết. Trong trường hợp các phần tử trong bảng ban đầu được đánh số từ 1 tới n và trường liên kết chính là số thứ tự của bản ghi trong bảng ban đầu như ở ví dụ trên, người ta gọi kỹ thuật này là kỹ thuật **sắp xếp bằng chỉ số**: Bảng ban đầu không hề bị ảnh hưởng gì cả, việc sắp xếp chỉ đơn thuần là đánh lại chỉ số cho các bản ghi theo thứ tự sắp xếp. Cụ thể hơn:

Nếu  $r[1..n]$  là các bản ghi cần sắp xếp theo một thứ tự nhất định thì việc sắp xếp bằng chỉ số tức là xây dựng một dãy  $Index[1..n]$  mà ở đây:

Index[j] = Chỉ số của bản ghi sẽ đứng thứ j khi sắp thứ tự  
(Bản ghi  $r[Index[j]]$  sẽ phải đứng sau  $j - 1$  bản ghi khác khi sắp xếp)

Do khoá có vai trò đặc biệt như vậy nên sau này, khi trình bày các giải thuật, ta sẽ coi **khoá như đại diện cho các bản ghi** và để cho đơn giản, ta chỉ nói tới giá trị của khoá mà thôi. Các thao tác trong kỹ thuật sắp xếp lẽ ra là tác động lên toàn bản ghi giờ đây chỉ làm trên khoá. Còn việc cài đặt các phương pháp sắp xếp trên danh sách các bản ghi và kỹ thuật sắp xếp bằng chỉ số, ta coi như bài tập.

### Bài toán sắp xếp giờ đây có thể phát biểu như sau:

Xét quan hệ thứ tự toàn phần “nhỏ hơn hoặc bằng” ký hiệu “ $\leq$ ” trên một tập hợp S, là quan hệ hai ngôi thoả mãn bốn tính chất:

Với  $\forall a, b, c \in S$

- ❖ Tính phẳng biến: Hoặc là  $a \leq b$ , hoặc  $b \leq a$ ;
- ❖ Tính phản xạ:  $a \leq a$
- ❖ Tính phản đối xứng: Nếu  $a \leq b$  và  $b \leq a$  thì bắt buộc  $a = b$ .
- ❖ Tính bắc cầu: Nếu có  $a \leq b$  và  $b \leq c$  thì  $a \leq c$ .

Trong trường hợp  $a \leq b$  và  $a \neq b$ , ta dùng ký hiệu “ $<$ ” cho gọn

Cho một dãy  $k[1..n]$  gồm n khoá. Giữa hai khoá bất kỳ có quan hệ thứ tự toàn phần “ $\leq$ ”. Xếp lại dãy các khoá đó để được dãy khoá thoả mãn  $k[1] \leq k[2] \leq \dots \leq k[n]$ .

Giả sử cấu trúc dữ liệu cho dãy khoá được mô tả như sau:

```
const
  n = ...; {Số khoá trong dãy khoá, có thể khai báo dưới dạng biến số nguyên để tùy biến hơn}
type
  TKey = ...; {Kiểu dữ liệu một khoá}
  TArray = array[1..n] of TKey;
var
  k: TArray; {Dãy khoá}
```

Thì những thuật toán sắp xếp dưới đây được viết dưới dạng thủ tục sắp xếp dãy khoá k, kiểu chỉ số đánh cho từng khoá trong dãy có thể coi là số nguyên Integer.

## 8.2. THUẬT TOÁN SẮP XẾP KIỂU CHỌN (SELECTIONSORT)

Một trong những thuật toán sắp xếp đơn giản nhất là phương pháp sắp xếp kiểu chọn. Ý tưởng cơ bản của cách sắp xếp này là:

Ở lượt thứ nhất, ta chọn trong dãy khoá  $k[1..n]$  ra khoá nhỏ nhất (khoá  $\leq$  mọi khoá khác) và đổi giá trị của nó với  $k[1]$ , khi đó giá trị khoá  $k[1]$  trở thành giá trị khoá nhỏ nhất.

Ở lượt thứ hai, ta chọn trong dãy khoá  $k[2..n]$  ra khoá nhỏ nhất và đổi giá trị của nó với  $k[2]$ .

...

Ở lượt thứ i, ta chọn trong dãy khoá  $k[i..n]$  ra khoá nhỏ nhất và đổi giá trị của nó với  $k[i]$ .

...

Làm tới lượt thứ  $n - 1$ , chọn trong hai khoá  $k[n-1], k[n]$  ra khoá nhỏ nhất và đổi giá trị của nó với  $k[n-1]$ .

```

procedure SelectionSort;
var
  i, j, jmin: Integer;
begin
  for i := 1 to n - 1 do {Làm n - 1 lượt}
  begin
    {Chọn trong số các khoá trong đoạn k[i..n] ra khoá k[jmin] nhỏ nhất}
    jmin := i;
    for j := i + 1 to n do
      if k[j] < k[jmin] then jmin := j;
    if jmin ≠ i then
      <Đào giá trị của k[jmin] cho k[i]>
    end;
  end;
end;

```

Đối với phương pháp kiểu lựa chọn, có thể coi phép so sánh ( $k[j] < k[jmin]$ ) là phép toán tích cực để đánh giá hiệu suất thuật toán về mặt thời gian. Ở lượt thứ  $i$ , để chọn ra khoá nhỏ nhất bao giờ cũng cần  $n - i$  phép so sánh, số lượng phép so sánh này không hề phụ thuộc gì vào tình trạng ban đầu của dãy khoá cả. Từ đó suy ra tổng số phép so sánh sẽ phải thực hiện là:

$$(n - 1) + (n - 2) + \dots + 1 = n * (n - 1) / 2$$

Vậy thuật toán sắp xếp kiểu chọn có độ phức tạp tính toán là  $O(n^2)$

### 8.3. THUẬT TOÁN SẮP XẾP NỐI BỌT (BUBBLESORT)

Trong thuật toán sắp xếp nối bọt, dãy các khoá sẽ được duyệt từ cuối dãy lên đầu dãy (từ  $k[n]$  về  $k[1]$ ), nếu gặp hai khoá kế cận bị ngược thứ tự thì đổi chỗ của chúng cho nhau. Sau lần duyệt như vậy, khoá nhỏ nhất trong dãy khoá sẽ được chuyển về vị trí đầu tiên và vấn đề trở thành sắp xếp dãy khoá từ  $k[2]$  tới  $k[n]$ :

```

procedure BubbleSort;
var
  i, j: Integer;
begin
  for i := 2 to n do
    for j := n downto i do {Duyệt từ cuối dãy lên, làm nổi khoá nhỏ nhất trong đoạn k[i-1, n] về vị trí i-1}
      if k[j] < k[j-1] then
        <Đào giá trị k[j] và k[j-1]>
end;

```

Đối với thuật toán sắp xếp nối bọt, có thể coi phép toán tích cực là phép so sánh  $k[j] < k[j-1]$ .

Và số lần thực hiện phép so sánh này là:

$$(n - 1) + (n - 2) + \dots + 1 = n * (n - 1) / 2$$

Vậy thuật toán sắp xếp nối bọt cũng có độ phức tạp là  $O(n^2)$ . Bất kể tình trạng dữ liệu vào như thế nào.

### 8.4. THUẬT TOÁN SẮP XẾP KIỂU CHÈN (INSERTIONSORT)

Xét dãy khoá  $k[1..n]$ . Ta thấy dãy con chỉ gồm mỗi một khoá là  $k[1]$  có thể coi là đã sắp xếp rồi. Xét thêm  $k[2]$ , ta so sánh nó với  $k[1]$ , nếu thấy  $k[2] < k[1]$  thì chèn nó vào trước  $k[1]$ . Đối với  $k[3]$ , ta lại xét dãy chỉ gồm 2 khoá  $k[1], k[2]$  đã sắp xếp và tìm cách chèn  $k[3]$  vào dãy khoá đó để được thứ tự sắp xếp. Một cách tổng quát, ta sẽ sắp xếp dãy  $k[1..i]$  trong điều kiện dãy  $k[1..i-1]$  đã sắp xếp rồi bằng cách chèn  $k[i]$  vào dãy đó tại vị trí đúng khi sắp xếp.

```

procedure InsertionSort;
var
  i, j: Integer;
  tmp: TKey; {Biên giữ lại giá trị khoá chèn}
begin
  for i := 2 to n do {Chèn giá trị k[i] vào dãy k[1..i-1] để toàn đoạn k[1..i] trở thành đã sắp xếp}
    begin
      tmp := k[i]; {Giữ lại giá trị k[i]}
      j := i - 1;
      while (j > 0) and (tmp < k[j]) do {So sánh giá trị cần chèn với lần lượt các khoá k[j] (i-1≥j≥0)}
        begin
          k[j+1] := k[j]; {Đẩy lùi giá trị k[j] về phía sau một vị trí, tạo ra "khoảng trống" tại vị trí j}
          j := j - 1;
        end;
      k[j+1] := tmp; {Đưa giá trị chèn vào "khoảng trống" mới tạo ra}
    end;
  end;
end;

```

Đối với thuật toán sắp xếp kiểu chèn, thì chi phí thời gian thực hiện thuật toán phụ thuộc vào tình trạng dãy khoá ban đầu. Nếu coi phép toán tích cực ở đây là phép so sánh  $tmp < k[j]$ , ta có:

Trường hợp tốt nhất ứng với dãy khoá đã sắp xếp rồi, mỗi lượt chỉ cần 1 phép so sánh, và như vậy tổng số phép so sánh được thực hiện là  $n - 1$ . Phân tích trong trường hợp tốt nhất, độ phức tạp tính toán của InsertionSort là  $\Theta(n)$

Trường hợp tồi tệ nhất ứng với dãy khoá đã có thứ tự ngược với thứ tự cần sắp thì ở lượt thứ  $i$ , cần có  $i - 1$  phép so sánh và tổng số phép so sánh là:

$$(n - 1) + (n - 2) + \dots + 1 = n * (n - 1) / 2.$$

Vậy phân tích trong trường hợp tốt nhất, độ phức tạp tính toán của InsertionSort là  $\Theta(n^2)$

Trường hợp các giá trị khoá xuất hiện một cách ngẫu nhiên, ta có thể coi xác suất xuất hiện mỗi khoá là đồng khả năng, thì có thể coi ở lượt thứ  $i$ , thuật toán cần trung bình  $i / 2$  phép so sánh và tổng số phép so sánh là:

$$(1 / 2) + (2 / 2) + \dots + (n / 2) = (n + 1) * n / 4.$$

Vậy phân tích trong trường hợp trung bình, độ phức tạp tính toán của InsertionSort là  $\Theta(n^2)$ .

Nhìn về kết quả đánh giá, ta có thể thấy rằng thuật toán sắp xếp kiểu chèn tỏ ra tốt hơn so với thuật toán sắp xếp chọn và sắp xếp nổi bọt. Tuy nhiên, chi phí thời gian thực hiện của thuật toán sắp xếp kiểu chèn vẫn còn khá lớn.

Có thể cải tiến thuật toán sắp xếp chèn nhờ nhận xét: Khi dãy khoá  $k[1..i-1]$  đã được sắp xếp thì việc tìm vị trí chèn có thể làm bằng thuật toán tìm kiếm nhị phân và kỹ thuật chèn có thể làm bằng các lệnh dịch chuyển vùng nhớ cho nhanh. Tuy nhiên điều đó cũng không làm giảm đi độ phức tạp của thuật toán bởi trong trường hợp xấu nhất, ta phải mất  $n - 1$  lần chèn và lần chèn thứ  $i$  ta phải dịch lùi  $i$  khoá để tạo ra khoảng trống trước khi đẩy giá trị khoá chèn vào chỗ trống đó.

```

procedure InsertionSortwithBinarySearching;
var
  i, inf, sup, median: Integer;
  tmp: TKey;
begin
  for i := 2 to n do
    begin
      tmp := k[i]; {Giữ lại giá trị k[i]}
      inf := 1; sup := i - 1; {Tìm chỗ chèn giá trị tmp vào đoạn từ k[inf] tới k[sup+1]}
      repeat {Sau mỗi vòng lặp này thì đoạn tìm bị co lại một nửa}
        median := (inf + sup) div 2; {Xét chỉ số nằm giữa chỉ số inf và chỉ số sup}
        if tmp < k[median] then sup := median - 1
        else inf := median + 1;
      until inf > sup; {Kết thúc vòng lặp thì inf = sup + 1 chính là vị trí chèn}
      <Địch các khoá từ k[inf] tới k[i-1] lùi sau một vị trí>
      k[inf] := tmp; {Đưa giá trị tmp vào "khoảng trống" mới tạo ra}
    end;
end;

```

## 8.5. SẮP XẾP CHÈN VỚI ĐỘ DÀI BUỚC GIẢM DẦN (SHELLSORT)

Nhược điểm của thuật toán sắp xếp kiểu chèn thể hiện khi mà ta luôn phải chèn một khóa vào vị trí gần đầu dãy. Để khắc phục nhược điểm này, người ta thường sử dụng thuật toán sắp xếp chèn với độ dài bước giảm dần, ý tưởng ban đầu cho thuật toán được đưa ra bởi D.L.Shell năm 1959 nên thuật toán còn có một tên gọi khác: ShellSort

Xét dãy khoá:  $k[1..n]$ . Với một số nguyên dương  $h$ :  $1 \leq h \leq n$ , ta có thể chia dãy đó thành  $h$  dãy con:

Dãy con 1:  $k[1], k[1+h], k[1 + 2h], \dots$

Dãy con 2:  $k[2], k[2+h], k[2 + 2h], \dots$

...

Dãy con  $h$ :  $k[h], k[2h], k[3h], \dots$

Ví dụ như dãy  $(4, 6, 7, 2, 3, 5, 1, 9, 8)$ ;  $n = 9$ ;  $h = 3$ . Có 3 dãy con.

| Dãy khoá chính: | 4 | 6 | 7 | 2 | 3 | 5 | 1 | 9 | 8 |
|-----------------|---|---|---|---|---|---|---|---|---|
| Dãy con 1:      | 4 |   |   | 2 |   |   | 1 |   |   |
| Dãy con 2:      |   | 6 |   |   | 3 |   |   | 9 |   |
| Dãy con 3:      |   |   | 7 |   |   | 5 |   |   | 8 |

Những dãy con như vậy được gọi là dãy con xếp theo độ dài bước  $h$ . Tư tưởng của thuật toán ShellSort là: Với một bước  $h$ , áp dụng thuật toán sắp xếp kiểu chèn từng dãy con độc lập để làm mịn dần dãy khoá chính. Rồi lại làm tương tự đối với bước  $h \text{ div } 2 \dots$  cho tới khi  $h = 1$  thì ta được dãy khoá sắp xếp.

Như ở ví dụ trên, nếu dùng thuật toán sắp xếp kiểu chèn thì khi gặp khoá  $k[7] = 1$ , là khoá nhỏ nhất trong dãy khoá, nó phải chèn vào vị trí 1, tức là phải thao tác trên 6 khoá đứng trước nó. Nhưng nếu coi 1 là khoá của dãy con 1 thì nó chỉ cần chèn vào trước 2 khoá trong dãy con đó

mà thôi. Đây chính là nguyên nhân ShellSort hiệu quả hơn sáp chèn: Khoá nhỏ được nhanh chóng đưa về **gần** vị trí đúng của nó.

```
procedure ShellSort;
var
  i, j, h: Integer;
  tmp: TKey;
begin
  begin
    h := n div 2;
    while h <> 0 do {Làm mìn dãy với độ dài bước h}
    begin
      for i := h + 1 to n do
        begin {Sắp xếp chèn trên dãy con a[i-h], a[i], a[i+h], a[i+2h], ...}
          tmp := k[i]; j := i - h;
          while (j > 0) and (k[j] > tmp) do
            begin
              k[j+h] := k[j];
              j := j - h;
            end;
          k[j+h] := tmp;
        end;
      h := h div 2;
    end;
  end;
```

Trên đây là phiên bản nguyên thuỷ của ShellSort do D.L.Shell đưa ra năm 1959. Độ dài bước được đếm div 2 sau mỗi lần lặp. Để thấy rằng để ShellSort hoạt động đúng thì chỉ cần dãy bước h giảm dần về 1 sau mỗi bước lặp là được, đã có một số nghiên cứu về việc chọn dãy bước h cho ShellSort nhằm tăng hiệu quả của thuật toán.

ShellSort hoạt động nhanh và dễ cài đặt, tuy vậy việc đánh giá độ phức tạp tính toán của ShellSort là tương đối khó, ta chỉ thừa nhận các kết quả sau đây:

Nếu các bước h được chọn theo thứ tự ngược từ dãy: 1, 3, 7, 15, ...,  $2^i - 1$ , ... thì độ phức tạp tính toán của ShellSort là  $O(n^{3/2})$ .

Nếu các bước h được chọn theo thứ tự ngược từ dãy: 1, 8, 23, 77, ...,  $4^{i+1} + 3 \cdot 2^i + 1$ , ... thì độ phức tạp tính toán của ShellSort là  $O(n^{4/3})$ .

Nếu các bước h được chọn theo thứ tự ngược từ dãy: 1, 2, 3, 4, 6, 8, 9, 12, 16, ...,  $2^i 3^j$ , ... (Dãy tăng dần của các phần tử dạng  $2^i 3^j$ ) thì độ phức tạp tính toán của ShellSort là  $O(n(\log n)^2)$ .

## 8.6. THUẬT TOÁN SẮP XẾP KIỀU PHÂN ĐOẠN (QUICKSORT)

### 8.6.1. Tư tưởng của QuickSort

QuickSort - thuật toán được đề xuất bởi C.A.R. Hoare - là một phương pháp sáp xếp tốt nhất, nghĩa là dù dãy khoá thuộc kiểu dữ liệu có thứ tự nào, QuickSort cũng có thể sáp xếp được và chưa có một thuật toán sáp xếp tổng quát nào nhanh hơn QuickSort về mặt tốc độ trung bình (theo tôi biết). Hoare đã mạnh dạn lấy chữ “Quick” để đặt tên cho thuật toán.

Ý tưởng chủ đạo của phương pháp có thể tóm tắt như sau: Sắp xếp dãy khoá  $k[1..n]$  thì có thể coi là sáp xếp đoạn từ chỉ số 1 tới chỉ số  $n$  trong dãy khoá đó. Để sáp xếp một đoạn trong dãy khoá, nếu đoạn đó có ít hơn 2 khoá thì không cần phải làm gì cả, còn nếu đoạn đó có ít nhất 2

khoá, ta chọn một khoá ngẫu nhiên nào đó của đoạn làm “chốt” (Pivot). Mọi khoá nhỏ hơn khoá chốt được xếp vào vị trí đứng trước chốt, mọi khoá lớn hơn khoá chốt được xếp vào vị trí đứng sau chốt. Sau phép hoán chuyển như vậy thì đoạn đang xét được chia làm hai đoạn khác rỗng mà mọi khoá trong đoạn đầu đều  $\leq$  chốt và mọi khoá trong đoạn sau đều  $\geq$  chốt. Hay nói cách khác: Mọi khoá trong đoạn đầu đều  $\leq$  mọi khoá trong đoạn sau. Và vấn đề trở thành sắp xếp hai đoạn mới tạo ra (có độ dài ngắn hơn đoạn ban đầu) bằng phương pháp tương tự.

```

procedure QuickSort;

procedure Partition(L, H: Integer); {Sắp xếp dãy khoá k[L..H]}
var
  i, j: Integer;
  Pivot: TKey; {Biến lưu giá trị khoá chốt}
begin
  if L >= H then Exit; {Nếu đoạn chỉ có 1 khoá thì không phải làm gì cả}
  Pivot := k[Random(H - L + 1) + L]; {Chọn một khoá ngẫu nhiên trong đoạn làm khoá chốt}
  i := L; j := H; {i := vị trí đầu đoạn; j := vị trí cuối đoạn}
  repeat
    while k[i] < Pivot do i := i + 1; {Tim từ đầu đoạn khoá ≥ khoá chốt}
    while k[j] > Pivot do j := j - 1; {Tim từ cuối đoạn khoá ≤ khoá chốt}
    {Đến đây ta tìm được hai khoá k[i] và k[j] mà k[i] ≥ key ≥ k[j]}
    if i ≤ j then
      begin
        if i < j then {Nếu chỉ số i đứng trước chỉ số j thì đảo giá trị hai khoá k[i] và k[j]}
          {Đảo giá trị k[i] và k[j]}; {Sau phép đảo này ta có: k[i] ≤ key ≤ k[j]}
        i := i + 1; j := j - 1;
      end;
    until i > j;
    Partition(L, j); Partition(i, H); {Sắp xếp hai đoạn con mới tạo ra}
  end;

begin
  Partition(1, n);
end;
```

Ta thử phân tích xem tại sao đoạn chương trình trên hoạt động đúng: Xét vòng lặp repeat...until trong lần lặp đầu tiên, vòng lặp while thứ nhất chắc chắn sẽ tìm được khoá  $k[i] \geq$  khoá chốt bởi chắc chắn tồn tại trong đoạn một khoá bằng khoá chốt. Tương tự như vậy, vòng lặp while thứ hai chắc chắn tìm được khoá  $k[j] \leq$  khoá chốt. Nếu như khoá  $k[i]$  đứng trước khoá  $k[j]$  thì ta đảo giá trị hai khoá, cho  $i$  tiến và  $j$  lùi. Khi đó ta có nhận xét rằng mọi khoá đứng trước vị trí  $i$  sẽ phải  $\leq$  khoá chốt và mọi khoá đứng sau vị trí  $j$  sẽ phải  $\geq$  khoá chốt.

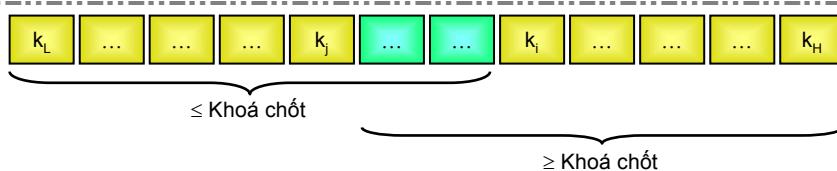


Hình 29: Vòng lặp trong của QuickSort

Điều này đảm bảo cho vòng lặp repeat...until tại bước sau, hai vòng lặp while...do bên trong chắc chắn lại tìm được hai khoá  $k[i]$  và  $k[j]$  mà  $k[i] \geq$  khoá chốt  $\geq k[j]$ , nếu khoá  $k[i]$  đứng trước khoá  $k[j]$  thì lại đảo giá trị của chúng, cho  $i$  tiến lên một vị trí và  $j$  lùi về một vị trí. Vậy vòng lặp repeat...until sẽ đảm bảo tại mỗi bước:

- ❖ Hai vòng lặp while...do bên trong luôn tìm được hai khoá  $k[i]$ ,  $k[j]$  mà  $k[i] \geq$  khoá chốt  $\geq k[j]$ . Không có trường hợp hai chỉ số  $i, j$  chạy ra ngoài đoạn (luôn luôn có  $L \leq i, j \leq H$ ).
- ❖ Sau mỗi phép hoán chuyển, mọi khoá đứng trước vị trí  $i$  luôn  $\leq$  khoá chốt và mọi khoá đứng sau vị trí  $j$  luôn  $\geq$  khoá chốt.

Vòng lặp repeat ...until sẽ kết thúc khi mà chỉ số  $i$  đứng phía sau chỉ số  $j$  (Hình 30).



**Hình 30: Trạng thái trước khi gọi đệ quy**

Theo những nhận xét trên, nếu có một khoá nằm giữa  $k[j]$  và  $k[i]$  thì khoá đó phải đúng bằng khoá chốt và nó đã được đặt đúng vị trí nên có thể bỏ qua khoá này mà chỉ xét hai đoạn ở hai đầu. Công việc còn lại là gọi đệ quy để làm tiếp với đoạn từ  $k[L]$  tới  $k[j]$  và đoạn từ  $k[i]$  tới  $k[H]$ . Hai đoạn này ngắn hơn đoạn đang xét bởi vì  $L \leq j < i \leq H$ . Vậy thuật toán không bao giờ bị rơi vào quá trình vô hạn mà sẽ dừng và cho kết quả đúng đắn.

Xét về độ phức tạp tính toán, trường hợp tốt nhất là tại mỗi bước chọn chốt để phân đoạn, ta chọn đúng trung vị của dãy khoá (giá trị sẽ đứng giữa dãy khi sắp thứ tự), khi đó độ phức tạp tính toán của QuickSort là  $\Theta(n \lg n)$ . Trường hợp tồi tệ nhất là tại mỗi bước chọn chốt để phân đoạn, ta chọn đúng vào khoá lớn nhất hoặc nhỏ nhất của dãy khoá, tạo ra một đoạn gồm 1 khoá và đoạn còn lại gồm  $n - 1$  khoá, khi đó độ phức tạp tính toán của QuickSort là  $\Theta(n^2)$ . Thời gian thực hiện giải thuật QuickSort trung bình là  $\Theta(n \lg n)$ . Việc chứng minh các kết quả này phải sử dụng những công cụ toán học phức tạp, ta thưa nhận những điều nói trên.

### 8.6.2. Trung vị và thứ tự thống kê (median and order statistics)

Việc chọn chốt cho phép phân đoạn quyết định hiệu quả của QuickSort, nếu chọn chốt không tốt, rất có thể việc phân đoạn bị suy biến thành trường hợp xấu khiến QuickSort hoạt động chậm và tràn ngắn xếp chương trình con khi gặp phải dây chuyền đệ qui quá dài. Những ví dụ sau đây cho thấy với một chiến lược chọn chốt tồi có thể dễ dàng tìm ra những bộ dữ liệu khiến QuickSort hoạt động chậm.

Với  $m$  khá lớn:

- ❖ Nếu như chọn chốt là khoá đầu đoạn ( $Pivot := k[L]$ ) hay chọn chốt là khoá cuối đoạn ( $Pivot := k[H]$ ) thì QuickSort sẽ trở thành “Slow” Sort với dãy  $(1, 2, \dots, m)$ .
- ❖ Nếu như chọn chốt là khoá giữa đoạn ( $Pivot := k[(L+H) \text{ div } 2]$ ) thì QuickSort cũng trở thành “Slow” Sort với dãy  $(1, 2, \dots, m-1, m, m, m-1, \dots, 2, 1)$ .
- ❖ Trong trường hợp chọn chốt là khoá nằm ở vị trí ngẫu nhiên trong đoạn, thật khó có thể tìm ra một bộ dữ liệu khiến cho QuickSort hoạt động chậm. Nhưng ta cũng cần hiểu rằng với mọi thuật toán tạo số ngẫu nhiên, trong  $m!$  dãy hoán vị của dãy  $(1, 2, \dots, m)$  thế nào cũng có một dãy làm QuickSort bị suy biến, tuy nhiên xác suất xảy ra dãy này quá nhỏ và

cũng rất khó để chỉ ra nên việc sử dụng cách chọn chốt là khoá nằm ở vị trí ngẫu nhiên có thể coi là an toàn với các trường hợp suy biến của QuickSort.

Phản “trung vị và thứ tự thống kê” này được trình bày trong nội dung thảo luận về QuickSort bởi nó cung cấp một chiến lược chọn chốt “đẹp” trên lý thuyết, nghĩa là trong trường hợp xấu nhất, độ phức tạp tính toán của QuickSort cũng chỉ là  $O(n \lg n)$  mà thôi. Để giải quyết vấn đề suy biến của QuickSort, ta xét bài toán tìm trung vị của dãy khoá và bài toán tổng quát hơn: Bài toán thứ tự thống kê (Order statistics).

**Bài toán:** Cho dãy khoá  $k_1, k_2, \dots, k_n$ , hãy chỉ ra khoá sẽ đúng thứ  $p$  trong dãy khi sắp thứ tự.

Khi  $p = n \div 2$  thì bài toán thứ tự thống kê trở thành bài toán tìm trung vị của dãy khoá. Sau đây ta sẽ nói về một số cách giải quyết bài toán thứ tự thống kê với mục tiêu cuối cùng là tìm ra một thuật toán để giải bài toán này với độ phức tạp trong trường hợp xấu nhất là  $O(n)$ .

Cách tệ nhất mà ai cũng có thể nghĩ tới là sắp xếp lại toàn bộ dãy  $k$  và đưa ra khoá đúng thứ  $p$  của dãy đã sắp. Trong các thuật toán sắp xếp tổng quát mà ta thảo luận trong bài, không thuật toán nào cho phép thực hiện việc này với độ phức tạp xấu nhất và trung bình là  $O(n)$  cả.

Cách thứ hai là sửa đổi một chút thủ tục Partition của QuickSort: thủ tục Partition chọn khoá chốt và chia đoạn đang xét làm hai đoạn con (thực ra là ba): Các khoá của đoạn đầu  $\leq$  chốt, các khoá của đoạn giữa = chốt, các khoá của đoạn sau  $\geq$  chốt. Khi đó ta hoàn toàn có thể xác định được khoá cần tìm nằm ở đoạn nào. Nếu khoá đó nằm ở đoạn giữa thì ta chỉ việc trả về giá trị khoá chốt. Nếu khoá đó nằm ở đoạn đầu hay đoạn sau thì chỉ cần gọi đệ quy làm tương tự với một trong hai đoạn đó chứ không cần gọi đệ quy để sắp xếp cả hai đoạn như QuickSort.

```
{
Input: Dãy khoá k[1..n], số p (1 ≤ p ≤ n)
Output: Giá trị khoá đúng thứ p trong dãy sau khi sắp thứ tự được trả về trong lời gọi hàm Select(1, n)
}
function Select(L, H: Integer): TKey; {Tìm trong đoạn k[L..H]}
var
  Pivot: TKey;
  i, j: Integer;
begin
  Pivot := k[Random(H - L + 1) + L];
  i := L; j := H;
  repeat
    while k[i] < Pivot do i := i + 1;
    while k[j] > Pivot do j := j - 1;
    if i ≤ j then
      begin
        if i < j then {Đảo giá trị k[i] và k[j]};
        i := i + 1; j := j - 1;
      end;
  until i > j;
  {Xác định khoá cần tìm nằm ở đoạn nào}
  if p ≤ j then Select := Select(L, j) {Khoá cần tìm nằm trong đoạn đầu}
  else
    if p ≥ i then Select := Select(i, H) {Khoá cần tìm nằm trong đoạn sau}
    else Select := Pivot; {Khoá cần tìm nằm ở đoạn giữa, chỉ cần trả về Pivot}
end;
```

Cách thứ hai tốt hơn cách thứ nhất khi phân tích độ phức tạp trung bình về thời gian thực hiện giải thuật (Có thể chứng minh được là  $O(n)$ ). Tuy nhiên trong trường hợp xấu nhất, giải

thuật này lại có độ phức tạp  $O(n^2)$  khi cần chỉ ra khoá lớn nhất của dãy khoá và chốt Pivot được chọn luôn là khoá nhỏ nhất của đoạn  $k[L..H]$ . Ta vẫn phải hướng tới một thuật toán tốt hơn nữa.

Cách thứ ba: Sự bí hiểm của số 5.

Ta sẽ viết một hàm  $Select(L, H, p)$  trả về khoá sẽ đứng thứ  $p$  khi sắp xếp dãy khoá  $k[L..H]$ . Nếu dãy này có ít hơn 50 khoá, thuật toán sắp xếp kiểu chèn sẽ được áp dụng trên dãy khoá này và sau đó giá trị  $k[L + p - 1]$  sẽ được trả về trong kết quả hàm  $Select$ .

Nếu dãy này có  $\geq 50$  khoá, ta chia các khoá  $k[L..H]$  thành các nhóm 5 khoá:

$k[L + 0..L + 4], k[L + 5..L + 9], k[L + 10, L + 14] \dots$

Nếu cuối cùng quá trình chia nhóm còn lại ít hơn 5 khoá (do độ dài đoạn  $k[L..H]$  không chia hết cho 5), ta bỏ qua không xét những khoá dư thừa này.

Với mỗi nhóm 5 khoá kể trên, ta tìm trung vị của nhóm (gọi tắt là trung vị nhóm - khoá đứng thứ 3 khi sắp thứ tự 5 khoá) và đẩy trung vị nhóm ra đầu đoạn  $k[L..H]$  theo thứ tự:

Trung vị của  $k[L + 0..L + 4]$  sẽ được đảo giá trị cho  $k[L]$

Trung vị của  $k[L + 5..L + 9]$  sẽ được đảo giá trị cho  $k[L + 1]$

...

Giả sử trung vị của nhóm cuối cùng sẽ được đảo giá trị cho  $k[j]$ .

Sau khi các trung vị nhóm đã tập trung về các vị trí  $k[L..j]$ , ta đặt Pivot bằng trung vị của các trung vị nhóm bằng một lệnh gọi đệ quy hàm  $Select$ :

$Pivot := Select(L, j, (j - L + 1) \text{ div } 2);$

Tiếp tục các lệnh của hàm  $Select$  như thế nào sẽ bàn sau, bây giờ ta giả sử hàm  $Select$  hoạt động đúng để xét một tính chất quan trọng của Pivot:

Nếu độ dài đoạn  $k[L..H]$  là  $\eta$  ( $= H - L + 1$ ) thì có  $\eta$  div 5 nhóm, nên cũng có  $\eta$  div 5 trung vị nhóm. Pivot là trung vị của các trung vị nhóm nên Pivot phải lớn hơn hay bằng ( $\eta$  div 5) div 2 trung vị nhóm, mỗi trung vị nhóm lại lớn hơn hay bằng 2 khoá khác của nhóm. Vậy có thể suy ra rằng Pivot lớn hơn hay bằng ( $\eta$  div 5 div 2 \* 3) khoá của đoạn  $k[L..H]$ . Lập luận tương tự, ta có Pivot nhỏ hơn hay bằng ( $\eta$  div 5 div 2 \* 3) khoá khác của đoạn  $k[L..H]$ . Với  $n \geq 50$ , ta có  $\eta$  div 5 div 2 \* 3  $\geq \eta/4$ . Suy ra:

- ❖ Có ít nhất  $\eta/4$  khoá nhỏ hơn hay bằng Pivot  $\Rightarrow$  có nhiều nhất  $3\eta/4$  khoá lớn hơn Pivot
- ❖ Có ít nhất  $\eta/4$  khoá lớn hơn hay bằng Pivot  $\Rightarrow$  có nhiều nhất  $3\eta/4$  khoá nhỏ hơn Pivot

Ta quay lại xây dựng tiếp hàm  $Select$ , khi đã có Pivot, ta có thể đếm được bao nhiêu khoá trong đoạn  $k[L..H]$  nhỏ hơn Pivot, bao nhiêu khoá bằng Pivot và bao nhiêu khoá lớn hơn Pivot, từ đó xác định được giá trị cần tìm nhỏ hơn, lớn hơn, hay bằng Pivot. Nếu giá trị cần tìm bằng Pivot thì chỉ cần trả về Pivot trong kết quả hàm. Nếu giá trị cần tìm nhỏ hơn Pivot, ta dồn tất cả các khoá nhỏ hơn Pivot trong đoạn  $k[L..H]$  về đầu đoạn và gọi đệ quy tìm tiếp với đoạn đầu này (Chú ý rằng độ dài đoạn được xét tiếp trong lời gọi đệ quy không quá  $3/4$  lần độ dài đoạn  $k[L..H]$ ), vẫn đè tương tự đối với trường hợp giá trị cần tìm lớn hơn Pivot.

```

procedure InsertionSort(L, H: Integer);
begin
  {Dùng InsertionSort sắp xếp dãy k[L..H];}
end;

function Select(L, H, p: Integer): TKey; {Hàm trả về khoá nhỏ thứ p trong dãy khoá k[L..H]}
var
  i, j, cL, cE: Integer;
  Pivot: TKey;
begin
  if H - L < 49 then {Nếu độ dài đoạn ít hơn 50 khoá}
    begin
      InsertionSort(L, H); {Thực hiện sắp xếp chèn}
      Select := k[L + p - 1]; {Và trả về phần tử nhỏ thứ p}
      Exit;
    end;
  j := L - 1; i := L;
  repeat {Tim trung vị của k[i, i + 4] chuyển về đầu đoạn}
    InsertionSort(i, i + 4);
    j := j + 1;
    {Đảo giá trị k[i + 2] cho k[j];}
    i := i + 5;
  until i + 5 > H;
  Pivot := Select(L, j, (j - L + 1) div 2);
  cL := 0; cE := 0; {đếm cL: số phần tử nhỏ hơn Pivot, cE: số phần tử bằng Pivot trong dãy k[L, H]}
  for i := L to H do
    if k[i] < Pivot then cL := cL + 1;
    else if k[i] = Pivot then cE := cE + 1;
  if (cL < p) and (p <= cL + cE) then {Giá trị cần tìm bằng Pivot}
    begin
      Select := Pivot;
      Exit;
    end;
  j := L - 1;
  if p <= cL then {Giá trị cần tìm nhỏ hơn Pivot}
    begin
      for i := L to H do {Đồn các khoá nhỏ hơn Pivot về đầu đoạn}
        if k[i] < Pivot then
          begin
            j := j + 1;
            {Đảo giá trị k[i] cho k[j];}
          end;
      Select := Select(L, j, p); {Gọi đệ quy tìm tiếp trong đoạn k[L..j]}
    end
  else {Giá trị cần tìm lớn hơn Pivot}
    begin
      for i := L to H do {Đồn các khoá lớn hơn Pivot về đầu đoạn}
        if k[i] > Pivot then
          begin
            j := j + 1;
            {Đảo giá trị k[i] cho k[j];}
          end;
      Select := Select(L, j, p - cL - cE); {Gọi đệ quy tìm tiếp trong đoạn k[L..j]}
    end;
  end;
end;

```

Ta sẽ chỉ ra rằng độ phức tạp tính toán của thuật toán trên là  $O(n)$  trong trường hợp xấu nhất. Nếu gọi  $T(n)$  là thời gian thực hiện hàm Select trong trường hợp xấu nhất với độ dài dãy khoá  $k[L..H]$  bằng  $n$ . Ta có:

$$T(n) \leq \begin{cases} c_1, & \text{if } n \leq 50 \\ c_2 n + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right), & \text{otherwise} \end{cases}$$

Bởi khi  $n \leq 50$  thì thuật toán sắp xếp chèn sẽ được thực hiện, có thể coi đoạn chương trình này kết thúc trong thời gian  $c_1$  với  $c_1$  là một hằng số đủ lớn. Khi  $n > 50$ , nhìn vào các đoạn mã trong hàm Select, lệnh Pivot := Select(L, j, (j - L + 1) div 2) có thời gian thực hiện  $T(n \text{ div } 5)$ . Lệnh Select := Select(L, j, ...) có thời gian thực hiện không quá  $T(3n/4)$  do tính chất của Pivot. Thời gian thực hiện các lệnh khác trong hàm Select tổng lại có thể coi là không quá  $c_2 \cdot n$  với  $c_2$  là một hằng số đủ lớn. Đặt  $c = \max(c_1, 20c_2)$ , ta có:

Với  $1 \leq n \leq 50$ , rõ ràng  $T(n) \leq c_1 \leq cn$ .

Với  $n > 50$ , giả thiết quy nạp rằng  $T(m) \leq cm$  với  $\forall m < n$ , ta sẽ chứng minh  $T(n) \leq cn$ , thật vậy:

$$T(n) \leq c_2 n + c \frac{n}{5} + c \frac{3n}{4} \leq c \frac{1}{20} n + c \frac{1}{5} n + c \frac{3}{4} n = cn$$

Ta có điều phải chứng minh:  $T(n) = O(n)$ . Sự bí ẩn của việc chọn số 5 cho kích thước nhóm đã được giải thích ( $1/5 + 3/4 < 1$ )

### 8.6.3. Kết luận:

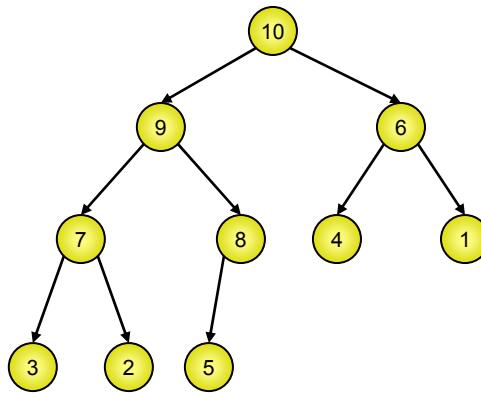
- ❖ Có thể giải bài toán thứ tự thống kê bằng thuật toán có độ phức tạp  $O(n)$  trong trường hợp xấu nhất.
- ❖ Có thể cài đặt thuật toán QuickSort với độ phức tạp  $O(nlgn)$  trong trường hợp xấu nhất bởi tại mỗi lần phân đoạn của QuickSort ta có thể tìm được trung vị của dãy trong thời gian  $O(n)$  bằng việc giải quyết bài toán thứ tự thống kê
- ❖ Cho tới thời điểm này, khi giải mọi bài toán có chứa thủ tục sắp xếp, ta có thể coi thời gian thực hiện thủ tục sắp xếp đó là  $O(nlgn)$  với mọi tình trạng dữ liệu vào.

## 8.7. THUẬT TOÁN SẮP XẾP KIỀU VŨNG ĐỒNG (HEAPSORT)

HeapSort được đề xuất bởi J.W.J. Williams năm 1981, thuật toán không những đóng góp một phương pháp sắp xếp hiệu quả mà còn xây dựng một cấu trúc dữ liệu quan trọng để biểu diễn hàng đợi có độ ưu tiên: Cấu trúc dữ liệu Heap.

### 8.7.1. Đồng (heap)

Đồng là một dạng cây nhị phân hoàn chỉnh đặc biệt mà giá trị lưu tại mọi nút có độ ưu tiên cao hơn hay bằng giá trị lưu trong hai nút con của nó. Trong thuật toán sắp xếp kiểu vùn đồng, ta coi quan hệ “ưu tiên hơn hay bằng” là quan hệ “lớn hơn hay bằng”:  $\geq$



Hình 31: Heap

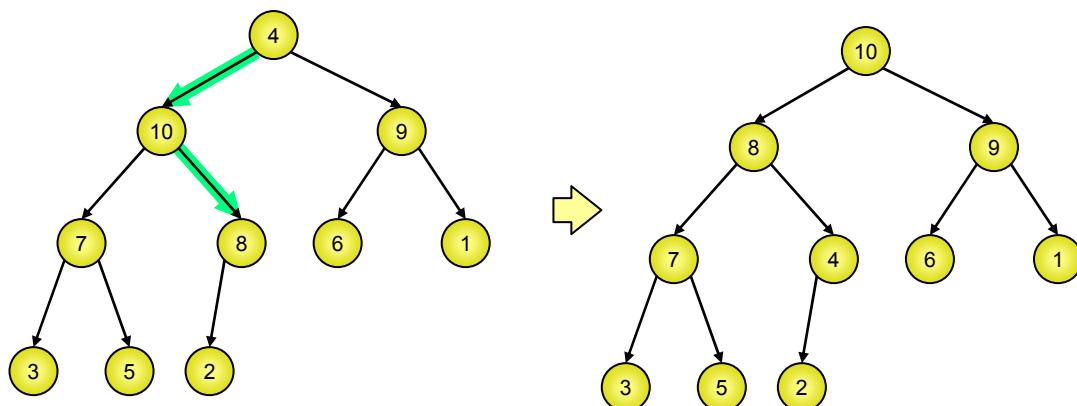
### 8.7.2. Vun đống

Trong bài §6, ta đã biết một dãy khoá  $k[1..n]$  là biểu diễn của một cây nhị phân hoàn chỉnh mà  $k[i]$  là giá trị lưu trong nút thứ  $i$ , nút con của nút thứ  $i$  là nút  $2i$  và nút  $2i + 1$ , nút cha của nút thứ  $j$  là nút  $j \text{ div } 2$ . Vấn đề đặt ra là sắp lại dãy khoá đã cho để nó biểu diễn một đống.

Vì cây nhị phân chỉ gồm có một nút hiển nhiên là đống, nên **để vun một nhánh cây gốc r thành đống, ta có thể coi hai nhánh con của nó (nhánh gốc  $2r$  và  $2r + 1$ ) đã là đống rồi** và thực hiện thuật toán vun đống từ dưới lên (bottom-up) đối với cây: Gọi  $h$  là chiều cao của cây, nút ở mức  $h$  (nút lá) đã là gốc một đống, ta vun lên để những nút ở mức  $h - 1$  cũng là gốc của đống, ... cứ như vậy cho tới nút ở mức 1 (nút gốc) cũng là gốc của đống.

**Thuật toán vun thành đống đối với cây gốc r, hai nhánh con của r đã là đống rồi:**

Giả sử ở nút  $r$  chứa giá trị  $V$ . Từ  $r$ , ta cứ đi tới nút con chứa giá trị lớn nhất trong 2 nút con, cho tới khi gặp phải một nút  $c$  mà mọi nút con của  $c$  đều chứa giá trị  $\leq V$  (nút lá cũng là trường hợp riêng của điều kiện này). Dọc trên đường đi từ  $r$  tới  $c$ , ta đẩy giá trị chứa ở nút con lên nút cha và đặt giá trị  $V$  vào nút  $c$ .



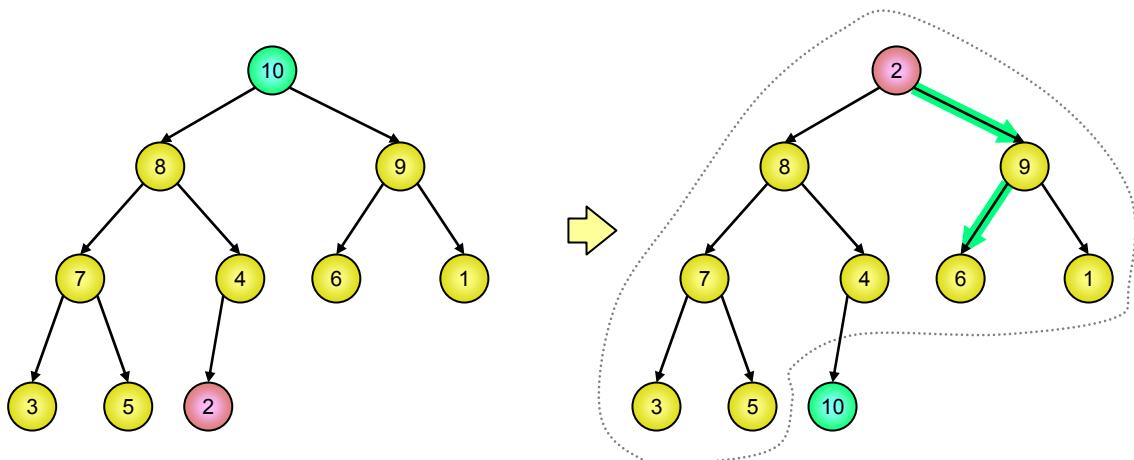
Hình 32: Vun đống

### 8.7.3. Tư tưởng của HeapSort

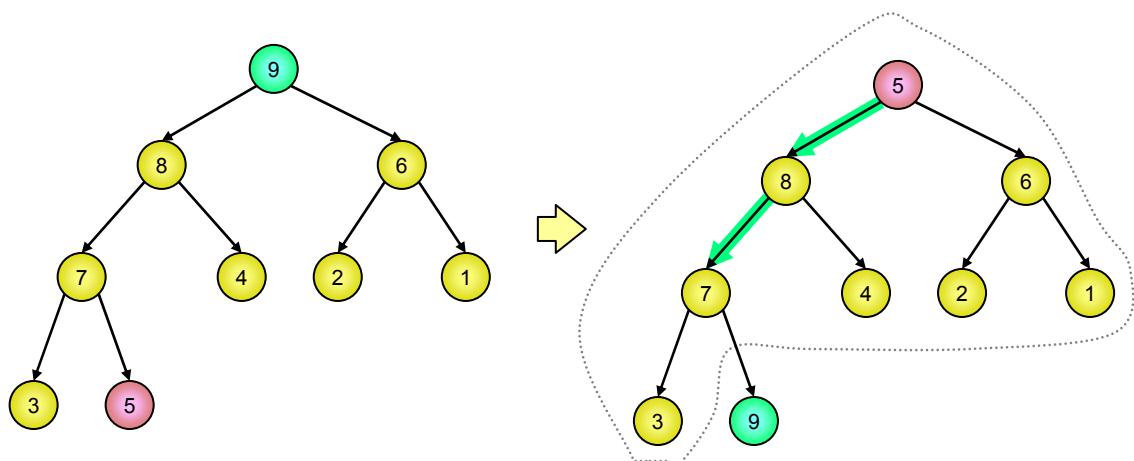
Đầu tiên, dãy khoá  $k[1..n]$  được vun từ dưới lên để nó biểu diễn một đống, khi đó khoá  $k[1]$  tương ứng với nút gốc của đống là khoá lớn nhất, ta đảo giá trị khoá đó cho  $k[n]$  và không tính tới  $k[n]$  nữa (Hình 33). Còn lại dãy khoá  $k[1..n-1]$  tuy không còn là biểu diễn của một

đóng nữa nhưng nó lại biểu diễn cây nhị phân hoàn chỉnh mà hai nhánh cây ở nút thứ 2 và nút thứ 3 (hai nút con của nút 1) đã là đóng rồi. Vậy chỉ cần vun một lần, ta lại được một đóng, đảo giá trị  $k[1]$  cho  $k[n-1]$  và tiếp tục cho tới khi đóng chỉ còn lại 1 nút (Hình 34).

Ví dụ:



Hình 33: Đảo giá trị  $k[1]$  cho  $k[n]$  và xét phần còn lại



Hình 34: Vun phần còn lại thành đóng rồi lại đảo giá trị  $k[1]$  cho  $k[n-1]$

Thuật toán HeapSort có hai thủ tục chính:

- ❖ Thủ tục Adjust(root, endnode) vun cây gốc root thành đóng trong điều kiện hai cây gốc  $2.root$  và  $2.root + 1$  đã là đóng rồi. Các nút từ  $endnode + 1$  tới  $n$  đã nằm ở vị trí đúng và không được tính tới nữa.
- ❖ Thủ tục HeapSort mô tả lại quá trình vun đóng và chọn khoá theo ý tưởng trên:

```

procedure HeapSort;
var
  r, i: Integer;

procedure Adjust(root, endnode: Integer); {Vun cây gốc Root thành đồng}
var
  c: Integer;
  Key: TKey; {Biến lưu giá trị khoá ở nút Root}
begin
  Key := k[root];
  while root * 2 ≤ endnode do {Chừng nào root chưa phải là lá}
    begin
      c := Root * 2; {Xét nút con trái của Root, so sánh với giá trị nút con phải, chọn ra nút mang giá trị lớn nhất}
      if (c < endnode) and (k[c] < k[c+1]) then c := c + 1;
      if k[c] ≤ Key then Break; {Cả hai nút con của Root đều mang giá trị ≤ Key thì dừng ngay}
      k[root] := k[c]; root := c; {Chuyển giá trị từ nút con c lên nút cha root và đi xuống xét nút con c}
    end;
  k[root] := Key; {Đặt giá trị Key vào nút root}
end;

begin {Bắt đầu thuật toán HeapSort}
  for r := n div 2 downto 1 do Adjust(r, n); {Vun cây từ dưới lên tạo thành đồng}
  for i := n downto 2 do
    begin
      {Đảo giá trị k[1] và k[i]; {Khoá lớn nhất được chuyển ra cuối dãy}
      Adjust(1, i - 1); {Vun phần còn lại thành đồng}
    end;
end;

```

Về độ phức tạp của thuật toán, ta đã biết rằng cây nhị phân hoàn chỉnh có  $n$  nút thì chiều cao của nó là  $\lceil \lg(n) \rceil + 1$ . Cứ cho là trong trường hợp xấu nhất thủ tục Adjust phải thực hiện tìm đường đi từ nút gốc tới nút lá ở xa nhất thì đường đi tìm được cũng chỉ dài bằng chiều cao của cây nên thời gian thực hiện một lần gọi Adjust là  $O(\lg n)$ . Từ đó có thể suy ra, trong trường hợp xấu nhất, độ phức tạp của HeapSort cũng chỉ là  $O(n \lg n)$ . Việc đánh giá thời gian thực hiện trung bình phức tạp hơn, ta chỉ ghi nhận một kết quả đã chứng minh được là độ phức tạp trung bình của HeapSort cũng là  $O(n \lg n)$ .

## 8.8. SẮP XẾP BẰNG PHÉP Đếm PHÂN PHỐI (DISTRIBUTION COUNTING)

Có một thuật toán sắp xếp đơn giản cho trường hợp đặc biệt: Dãy khoá  $k[1..n]$  là các số nguyên nằm trong khoảng từ 0 tới  $M$  ( $TKey = 0..M$ ).

Ta dựng dãy  $c[0..M]$  các biến đếm, ở đây  $c[V]$  là số lần xuất hiện giá trị  $V$  trong dãy khoá:

```

for V := 0 to M do c[V] := 0; {Khởi tạo dãy biến đếm}
for i := 1 to n do c[k[i]] := c[k[i]] + 1;

```

Ví dụ với dãy khoá: 1, 2, 2, 3, 0, 0, 1, 1, 3, 3 ( $n = 10$ ,  $M = 3$ ), sau bước đếm ta có:

$c[0] = 2$ ;  $c[1] = 3$ ;  $c[2] = 2$ ;  $c[3] = 3$ .

Dựa vào dãy biến đếm, ta hoàn toàn có thể biết được: sau khi sắp xếp thì giá trị  $V$  phải nằm từ vị trí nào tới vị trí nào. Như ví dụ trên thì giá trị 0 phải nằm từ vị trí 1 tới vị trí 2; giá trị 1 phải đứng liên tiếp từ vị trí 3 tới vị trí 5; giá trị 2 đứng ở vị trí 6 và 7 còn giá trị 3 nằm ở ba vị trí cuối 8, 9, 10:

0 0 1 1 2 2 3 3 3

Tức là sau khi sắp xếp:

Giá trị 0 đứng trong đoạn từ vị trí 1 tới vị trí c[0].

Giá trị 1 đứng trong đoạn từ vị trí c[0]+1 tới vị trí c[0]+c[1].

Giá trị 2 đứng trong đoạn từ vị trí c[0]+c[1]+1 tới vị trí c[0]+c[1] c[2].

...

Giá trị v trong đoạn đứng từ vị trí c[0]+... +c[v-1]+1 tới vị trí c[0]+...+ c[v].

...

Để ý vị trí cuối của mỗi đoạn, nếu ta tính lại dãy c như sau:

```
for V := 1 to M do c[V] := c[V-1] + c[V]
```

Thì **c[V]** là vị trí cuối của đoạn chứa giá trị V trong dãy khoá đã sắp xếp.

Muốn dựng lại dãy khoá sắp xếp, ta thêm một dãy khoá phụ x[1..n]. Sau đó duyệt lại dãy khoá k, mỗi khi gặp khoá mang giá trị V ta đưa giá trị đó vào khoá x[c[V]] và giảm c[V] đi 1.

```
for i := n downto 1 do
begin
  V := k[i];
  X[c[V]] := k[i]; c[V] := c[V] - 1;
end;
```

Khi đó dãy khoá x chính là dãy khoá đã được sắp xếp, công việc cuối cùng là gán giá trị dãy khoá x cho dãy khoá k.

```
procedure DistributionCounting; { TKey = 0..M }
var
  c: array[0..M] of Integer; { Dãy biến đếm số lần xuất hiện mỗi giá trị }
  t: TArray; { Dãy khoá phụ }
  i: Integer;
  V: TKey;
begin
  for V := 0 to M do c[V] := 0; { Khởi tạo dãy biến đếm }
  for i := 1 to n do c[k[i]] := c[k[i]] + 1; { Đếm số lần xuất hiện các giá trị }
  for V := 1 to M do c[V] := c[V-1] + c[V]; { Tính vị trí cuối mỗi đoạn }
  for i := n downto 1 do
    begin
      V := k[i];
      t[c[V]] := k[i]; c[V] := c[V] - 1;
    end;
  k := x; { Sao chép giá trị từ dãy khoá x sang dãy khoá k }
end;
```

Rõ ràng độ phức tạp của phép đếm phân phối là  $O(M + n)$ . Nhược điểm của phép đếm phân phối là khi tập giá trị khoá quá lớn thì cho dù n nhỏ cũng không thể làm được.

Có thể có thắc mắc tại sao trong thao tác dựng dãy khoá t, phép duyệt dãy khoá k theo thứ tự nào thì kết quả sắp xếp cũng vẫn đúng, vậy tại sao ta lại chọn phép duyệt ngược từ dưới lên?.

Để trả lời câu hỏi này, ta phải phân tích thêm một đặc trưng của các thuật toán sắp xếp:

## 8.9. TÍNH ỔN ĐỊNH CỦA THUẬT TOÁN SẮP XẾP (STABILITY)

Một phương pháp sắp xếp được gọi là **ổn định** nếu nó bảo toàn thứ tự ban đầu của các bản ghi mang khoá bằng nhau trong danh sách. Ví dụ như ban đầu danh sách sinh viên được xếp theo thứ tự tên alphabet, thì khi sắp xếp danh sách sinh viên theo thứ tự giảm dần của điểm thi,

những sinh viên bằng nhau sẽ được dồn về một đoạn trong danh sách và vẫn được giữ nguyên thứ tự tên alphabet.

Hãy xem lại những thuật toán sắp xếp ở trước, trong những thuật toán đó, thuật toán sắp xếp nổi bọt, thuật toán sắp xếp chèn và phép đếm phân phối là những thuật toán sắp xếp ổn định, còn những thuật toán sắp xếp khác (và nói chung những thuật toán sắp xếp đòi hỏi phải đảo giá trị 2 bản ghi ở vị trí bất kỳ) là không ổn định.

Với phép đếm phân phối ở mục trước, ta nhận xét rằng nếu hai bản ghi có khoá sắp xếp bằng nhau thì khi đưa giá trị vào dãy bản ghi phụ, bản ghi nào vào trước sẽ nằm phía sau. Vậy nên ta sẽ đẩy giá trị các bản ghi vào dãy phụ theo thứ tự ngược để giữ được thứ tự tương đối ban đầu.

Nói chung, mọi phương pháp sắp xếp tổng quát cho dù không ổn định thì đều có thể biến đổi để nó trở thành ổn định, phương pháp chung nhất được thể hiện qua ví dụ sau:

Giả sử ta cần sắp xếp các sinh viên trong danh sách theo thứ tự giảm dần của điểm bằng một thuật toán sắp xếp ổn định. Ta thêm cho mỗi sinh viên một khoá Index là thứ tự ban đầu của anh ta trong danh sách. Trong thuật toán sắp xếp được áp dụng, cứ chỗ nào cần so sánh hai sinh viên A và B xem ai phải đứng trước, trước hết ta quan tâm tới điểm số: Nếu điểm của A khác điểm của B thì người nào điểm cao hơn sẽ đứng trước, nếu điểm số bằng nhau thì người nào có Index nhỏ hơn sẽ đứng trước.

Trong một số bài toán, tính ổn định của thuật toán sắp xếp quyết định tới cả tính đúng đắn của toàn thuật toán lớn. Chính tính “nhanh” của QuickSort và tính ổn định của phép đếm phân phối là cơ sở nền tảng cho hai thuật toán sắp xếp cực nhanh trên các dãy khoá số mà ta sẽ trình bày dưới đây.

## 8.10. THUẬT TOÁN SẮP XẾP BẰNG CƠ SỐ (RADIX SORT)

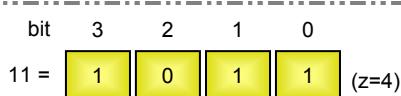
Bài toán đặt ra là: Cho dãy khoá là các số tự nhiên  $k[1..n]$  hãy sắp xếp chúng theo thứ tự không giảm. (Trong trường hợp ta đang xét, TKey là kiểu số tự nhiên)

### 8.10.1. Sắp xếp cơ số theo kiểu hoán vị các khoá (Radix Exchange Sort)

Hãy xem lại thuật toán QuickSort, tại bước phân đoạn nó phân đoạn đang xét thành hai đoạn thoả mãn mỗi khoá trong đoạn đầu  $\leq$  mọi khoá trong đoạn sau và thực hiện tương tự trên hai đoạn mới tạo ra, việc phân đoạn được tiến hành với sự so sánh các khoá với giá trị một khoá chốt.

Đối với các số nguyên thì ta có thể coi mỗi số nguyên là một dãy  $z$  bit đánh số từ bit 0 (bit ở hàng đơn vị) tới bit  $z - 1$  (bit cao nhất).

Ví dụ:



**Hình 35: Đánh số các bit**

Vậy thì tại bước phân đoạn dãy khoá từ  $k[1]$  tới  $k[n]$ , ta có thể đưa những khoá có bit cao nhất là 0 về đầu dãy, những khoá có bit cao nhất là 1 về cuối dãy. Để thấy rằng những khoá bắt đầu bằng bit 0 sẽ phải nhỏ hơn những khoá bắt đầu bằng bit 1. Tiếp tục quá trình phân đoạn với hai đoạn dãy khoá: Đoạn gồm các khoá có bit cao nhất là 0 và đoạn gồm các khoá có bit cao nhất là 1. Với những khoá thuộc cùng một đoạn thì có bit cao nhất giống nhau, nên ta có thể áp dụng quá trình phân đoạn tương tự trên theo bit thứ  $z - 2$  và cứ tiếp tục như vậy ...

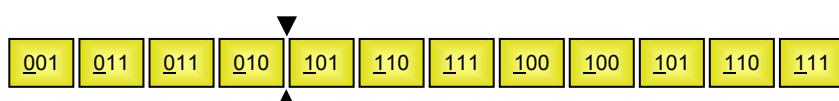
Quá trình phân đoạn kết thúc nếu như đoạn đang xét là rỗng hay ta đã tiến hành phân đoạn đến tận bit đơn vị, tức là tất cả các khoá thuộc một trong hai đoạn mới tạo ra đều có bit đơn vị bằng nhau (điều này đồng nghĩa với sự bằng nhau ở tất cả những bit khác, tức là bằng nhau về giá trị khoá).

Ví dụ:

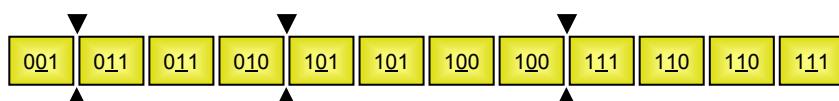
Xét dãy khoá: 1, 3, 7, 6, 5, 2, 3, 4, 4, 5, 6, 7. Tương ứng với các dãy 3 bit:



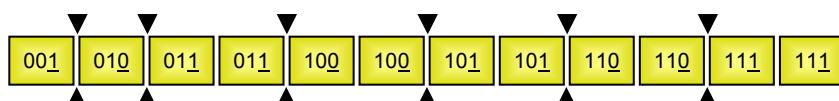
Trước hết ta chia đoạn dựa vào bit 2 (bit cao nhất):



Sau đó chia tiếp hai đoạn tạo ra dựa vào bit 1:



Cuối cùng, chia tiếp những đoạn tạo ra dựa vào bit 0:



Ta được dãy khoá tương ứng: 1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7 là dãy khoá sắp xếp.

Quá trình chia đoạn dựa vào bit b có thể chia thành một đoạn rỗng và một đoạn gồm toàn bộ các khoá còn lại, nhưng việc chia đoạn không bao giờ bị rời vào quá trình đệ quy vô hạn bởi những lần đệ quy tiếp theo sẽ phân đoạn dựa vào bit  $b - 1$ ,  $b - 2$  ... và sẽ phải dừng lại khi xét tới bit 0. Công việc còn lại là cố gắng hiểu đoạn chương trình sau và phân tích xem tại sao nó hoạt động đúng:

```

procedure RadixExchangeSort;
var
  z: Integer; {Độ dài dãy bit biểu diễn mỗi khoá}

procedure Partition(L, H, b: Integer); {Phân đoạn [L, H] dựa vào bit b}
var
  i, j: Integer;
begin
  if L ≥ H then Exit;
  i := L; j := H;
  repeat
    {Hai vòng lặp trong dưới đây luôn cầm canh i < j}
    while (i < j) and (Bit b của k[i] = 0) do i := i + 1; {Tìm khoá có bit b = 1 từ đầu đoạn}
    while (i < j) and (Bit b của k[j] = 1) do j := j - 1; {Tìm khoá có bit b = 0 từ cuối đoạn}
    {Đảo giá trị k[i] cho k[j];}
  until i = j;
  if <Bit b của k[j] = 0> then j := j + 1; {j là điểm bắt đầu của đoạn có bit b là 1}
  if b > 0 then {Chưa xét tới bit đơn vị}
    begin
      Partition(L, j - 1, b - 1); Partition(j, R, b - 1);
    end;
  end;
end;

begin
  {Dựa vào giá trị lớn nhất của dãy khoá, xác định z là độ dài dãy bit biểu diễn mỗi khoá};
  Partition(1, n, z - 1);
end;

```

Với Radix Exchange Sort, ta hoàn toàn có thể làm trên hệ cơ số R khác chứ không nhất thiết phải làm trên hệ nhị phân (ý tưởng cũng tương tự như trên), tuy nhiên quá trình phân đoạn sẽ không phải chia làm 2 mà chia thành R đoạn. Về độ phức tạp của thuật toán, ta thấy để phân đoạn bằng một bit thì thời gian sẽ là  $C \cdot n$  để chia tất cả các đoạn cần chia bằng bit đó ( $C$  là hằng số). Vậy tổng thời gian phân đoạn bằng  $z$  bit sẽ là  $C \cdot n \cdot z$ . Trong trường hợp xấu nhất, độ phức tạp của Radix Exchange Sort là  $O(n \cdot z)$ . Và độ phức tạp trung bình của Radix Exchange Sort là  $O(n \cdot \min(z, \lg n))$ .

Nói chung, Radix Exchange Sort cài đặt như trên chỉ thể hiện tốc độ tối đa trên các hệ thống cho phép xử lý trực tiếp trên các bit: Hệ thống phải cho phép lấy một bit ra dễ dàng và thao tác với thời gian nhanh hơn hẳn so với thao tác trên BYTE, WORD, DWORD, QWORD... Khi đó Radix Exchange Sort sẽ tốt hơn nhiều QuickSort. (Ta thử lập trình sắp xếp các dãy nhị phân độ dài  $z$  theo thứ tự từ điển để khảo sát). Trên các máy tính hiện nay chỉ cho phép xử lý trực tiếp trên BYTE (hay WORD, DWORD v.v...), việc tách một bit ra khỏi Byte đó để xử lý lại rất chậm và làm ảnh hưởng không nhỏ tới tốc độ của Radix Exchange Sort. Chính vì vậy, tuy đây là một phương pháp hay, nhưng khi cài đặt cụ thể thì tốc độ cũng chỉ ngang ngửa chứ không thể qua mặt QuickSort được.

### 8.10.2. Sắp xếp cơ số trực tiếp (Straight Radix Sort)

Ta sẽ trình bày ý tưởng chính của phương pháp sắp xếp cơ số trực tiếp bằng một ví dụ: Sắp xếp dãy khoá:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 925 | 817 | 821 | 638 | 639 | 744 | 742 | 563 | 570 | 166 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Trước hết, ta sắp xếp dãy khoá này theo thứ tự tăng dần của chữ số hàng đơn vị bằng một thuật toán sắp xếp khác, được dãy khoá:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 570 | 821 | 742 | 563 | 744 | 925 | 166 | 817 | 638 | 639 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Sau đó, ta sắp xếp dãy khoá mới tạo thành theo thứ tự tăng dần của chữ số hàng chục bằng một thuật toán sắp xếp  **ổn định**, được dãy khoá:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 817 | 821 | 925 | 638 | 639 | 742 | 744 | 563 | 166 | 570 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Vì thuật toán sắp xếp ta sử dụng là ổn định, nên nếu hai khoá có chữ số hàng chục giống nhau thì khoá nào có chữ số hàng đơn vị nhỏ hơn sẽ đứng trước. Nói như vậy có nghĩa là dãy khoá thu được sẽ có thứ tự tăng dần về giá trị tạo thành từ hai chữ số cuối.

Cuối cùng, ta sắp xếp lại dãy khoá theo thứ tự tăng dần của chữ số hàng trăm cũng bằng một thuật toán sắp xếp ổn định, thu được dãy khoá:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 166 | 563 | 570 | 638 | 639 | 742 | 744 | 817 | 821 | 925 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Lập luận tương tự như trên dựa vào tính ổn định của phép sắp xếp, dãy khoá thu được sẽ có thứ tự tăng dần về giá trị tạo thành bởi cả ba chữ số, đó là dãy khoá đã sắp.

### Nhận xét:

Ta hoàn toàn có thể coi số chữ số của mỗi khoá là bằng nhau, như ví dụ trên nếu có số 15 trong dãy khoá thì ta có thể coi nó là 015.

Cũng từ ví dụ, ta có thể thấy rằng số lượt thao tác sắp xếp phải áp dụng đúng bằng số chữ số tạo thành một khoá. Với một hệ cơ số lớn, biểu diễn một giá trị khoá sẽ phải dùng ít chữ số hơn. Ví dụ số 12345 trong hệ thập phân phải dùng tới 5 chữ số, còn trong hệ cơ số 1000 chỉ cần dùng 2 chữ số AB mà thôi, ở đây A là chữ số mang giá trị 12 còn B là chữ số mang giá trị 345.

Tốc độ của sắp xếp cơ số trực tiếp phụ thuộc rất nhiều vào thuật toán sắp xếp ổn định tại mỗi bước. Không có một lựa chọn nào khác tốt hơn phép đếm phân phôi. Tuy nhiên, phép đếm phân phôi có thể không cài đặt được hoặc kém hiệu quả nếu như tập giá trị khoá quá rộng, không cho phép dựng ra dãy các biến đếm hoặc phải sử dụng dãy biến đếm quá dài (Điều này xảy ra nếu chọn hệ cơ số quá lớn).

Một lựa chọn khôn ngoan là nên chọn hệ cơ số thích hợp cho từng trường hợp cụ thể để dung hoà tới mức tối ưu nhất ba mục tiêu:

- ❖ Việc lấy ra một chữ số của một số được thực hiện dễ dàng
- ❖ Sử dụng ít lần gọi phép đếm phân phôi.
- ❖ Phép đếm phân phôi thực hiện nhanh

```

procedure StraightRadixSort;
const
  radix = ...; {Tuỳ chọn hệ cơ số radix cho hợp lý}
var
  t: TArray; {Dãy khoá phụ}
  p: Integer;
  nDigit: Integer; {Số chữ số cho một khoá, đánh số từ chữ số thứ 0 là hàng đơn vị đến chữ số thứ nDigit - 1}
  Flag: Boolean; {Flag = True thì sắp dãy k, ghi kết quả vào dãy t; Flag = False thì sắp dãy t, ghi kq vào k}

function GetDigit(Num: TKey; p: Integer): Integer; {Lấy chữ số thứ p của số Num (0≤p<nDigit)}
begin
  GetDigit := Num div radixp mod radix; {Trường hợp cụ thể có thể có cách viết tốt hơn}
end;

{Sắp xếp ổn định dãy số x theo thứ tự tăng dần của chữ số thứ p, kết quả sắp xếp được chia vào dãy số y}
procedure DCount(var x, y: TArray; p: Integer); {Thuật toán đếm phân phối, sắp từ x sang y}
var
  c: array[0..radix - 1] of Integer; {c[d] là số lần xuất hiện chữ số d tại vị trí p}
  i, d: Integer;
begin
  for d := 0 to radix - 1 do c[d] := 0;
  for i := 1 to n do
    begin
      d := GetDigit(x[i], p); c[d] := c[d] + 1;
    end;
  for d := 1 to radix - 1 do c[d] := c[d-1] + c[d]; {các c[d] trở thành các mốc cuối đoạn}
  for i := n downto 1 do {Điền giá trị vào dãy y}
    begin
      d := GetDigit(x[i], p);
      y[c[d]] := x[i]; c[d] := c[d] - 1;
    end;
  end;

begin {Thuật toán sắp xếp cơ số trực tiếp}
  {Dựa vào giá trị lớn nhất trong dãy khoá, xác định nDigit là số chữ số phải dùng cho mỗi khoá trong hệ radix};
  Flag := True;
  for p := 0 to nDigit - 1 do {Xét từ chữ số hàng đơn vị lên, sắp xếp ổn định theo chữ số thứ p}
    begin
      if Flag then DCount(k, t, p) else DCount(t, k, p);
      Flag := not Flag; {Đảo cờ, dùng k tính t rồi lại dùng t tính k ...}
    end;
  if not Flag then k := t; {Nếu kết quả cuối cùng đang ở trong t thì sao chép giá trị từ t sang k}
end;

```

Xét phép đếm phân phối, ta đã biết độ phức tạp của nó là  $O(\max(\text{radix}, n))$ . Mà radix là một hằng số tự ta chọn từ trước, nên khi n lớn, độ phức tạp của phép đếm phân phối là  $O(n)$ . Thuật toán sử dụng nDigit lần phép đếm phân phối nên có thể thấy độ phức tạp của thuật toán là  $O(n \cdot nDigit)$  bất kể dữ liệu đầu vào.

Ta có thể coi sắp xếp cơ số trực tiếp là một mở rộng của phép đếm phân phối, khi dãy số chỉ toàn các số có 1 chữ số (trong hệ radix) thì đó chính là phép đếm phân phối. Sự khác biệt ở đây là: Sắp xếp cơ số trực tiếp có thể thực hiện với các khoá mang giá trị lớn; còn phép đếm phân phối chỉ có thể làm trong trường hợp các khoá mang giá trị nhỏ, bởi nó cần một lượng bộ nhớ đủ rộng để giăng ra dãy biến đếm số lần xuất hiện cho từng giá trị.

## 8.11. THUẬT TOÁN SẮP XẾP TRỘN (MERGESORT)

Thuật toán sắp xếp trộn (MergeSort hay Collation Sort) là một trong những thuật toán sắp xếp cổ điển nhất, được đề xuất bởi J.von Neumann năm 1945. Cho tới nay, người ta vẫn coi MergeSort là một thuật toán sắp xếp ngoài mực, được đưa vào giảng dạy rộng rãi và được tích hợp trong nhiều phần mềm thương mại.

### 8.11.1. Phép trộn 2 đường

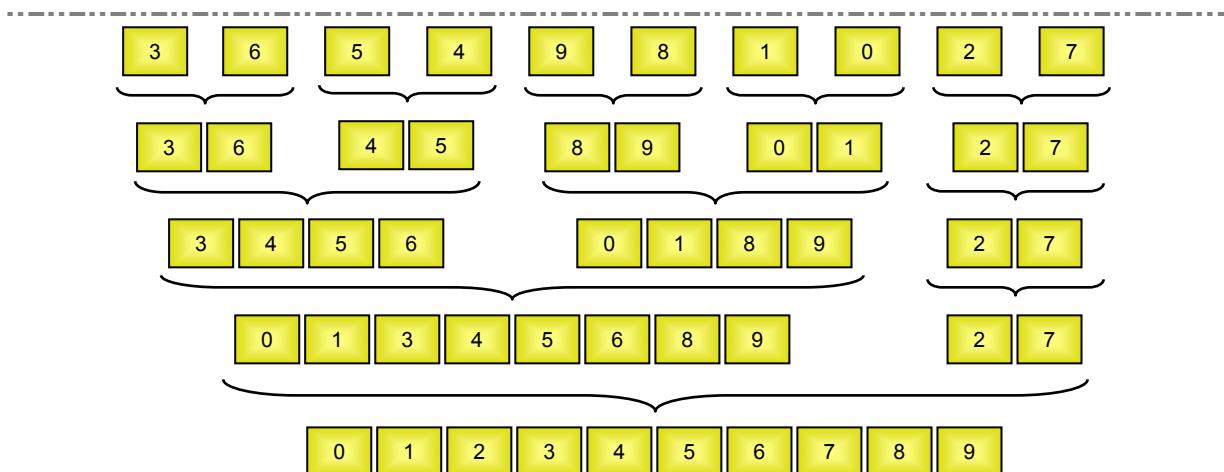
Phép trộn 2 đường là phép hợp nhất hai dãy khoá **đã sắp xếp** để ghép lại thành một dãy khoá có kích thước bằng tổng kích thước của hai dãy khoá ban đầu và dãy khoá tạo thành cũng có thứ tự sắp xếp. Nguyên tắc thực hiện của nó khá đơn giản: so sánh hai khoá đứng đầu hai dãy, chọn ra khoá nhỏ nhất và đưa nó vào miền sắp xếp (một dãy khoá phụ có kích thước bằng tổng kích thước hai dãy khoá ban đầu) ở vị trí thích hợp. Sau đó, khoá này bị loại ra khỏi dãy khoá chứa nó. Quá trình tiếp tục cho tới khi một trong hai dãy khoá đã cạn, khi đó chỉ cần chuyển toàn bộ dãy khoá còn lại ra miền sắp xếp là xong.

Ví dụ: Với hai dãy khoá: (1, 3, 10, 11) và (2, 4, 9)

| Dãy 1          | Dãy 2     | Khoá nhỏ nhất trong 2 dãy                  | Miền sắp xếp            |
|----------------|-----------|--|-------------------------|
| (1, 3, 10, 11) | (2, 4, 9) | 1  | (1)                     |
| (3, 10, 11)    | (2, 4, 9) | 2  | (1, 2)                  |
| (3, 10, 11)    | (4, 9)    | 3  | (1, 2, 3)               |
| (10, 11)       | (4, 9)    | 4  | (1, 2, 3, 4)            |
| (10, 11)       | (9)       | 9  | (1, 2, 3, 4, 9)         |
| (10, 11)       | Ø         | Dãy 2 là Ø, đưa nốt dãy 1 vào miền sắp xếp | (1, 2, 3, 4, 9, 10, 11) |

### 8.11.2. Sắp xếp bằng trộn 2 đường trực tiếp

Ta có thể coi mỗi khoá trong dãy khoá  $k[1..n]$  là một mạch với độ dài 1, dĩ nhiên các mạch độ dài 1 có thể coi là **đã** được sắp. Nếu trộn hai mạch liên tiếp lại thành một mạch có độ dài 2, ta lại được dãy gồm các mạch **đã** được sắp. Cứ tiếp tục như vậy, số mạch trong dãy sẽ giảm dần sau mỗi lần trộn (Hình 36)



Hình 36: Thuật toán sắp xếp trộn

Để tiến hành thuật toán sắp xếp trộn hai đường trực tiếp, ta viết các thủ tục:

Thủ tục Merge(var x, y: TArray; a, b, c: Integer); thủ tục này trộn mảnh x[a..b] với mảnh x[b+1..c] để được mảnh y[a..c].

Thủ tục MergeByLength(var x, y: TArray; len: Integer); thủ tục này trộn lần lượt các cặp mảnh theo thứ tự:

Trộn mảnh x[1..len] và x[len+1..2len] thành mảnh y[1..2len].

Trộn mảnh x[2len+1..3len] và x[3len+1..4len] thành mảnh y[2len+1..4len].

...

Lưu ý rằng đến cuối cùng ta có thể gặp hai trường hợp: Hoặc còn lại hai mảnh mà mảnh thứ hai có độ dài  $< len$ . Hoặc chỉ còn lại một mảnh. Trường hợp thứ nhất ta phải quản lý chính xác các chỉ số để thực hiện phép trộn, còn trường hợp thứ hai thì không được quên thao tác đưa thẳng mảnh duy nhất còn lại sang dãy y.

Cuối cùng là thủ tục MergeSort, thủ tục này cần một dãy khoá phụ t[1..n]. Trước hết ta gọi MergeByLength(k, t, 1) để trộn hai khoá liên tiếp của k thành một mảnh trong t, sau đó lại gọi MergeByLength(t, k, 2) để trộn hai mảnh liên tiếp trong t thành một mảnh trong k, rồi lại gọi MergeByLength(k, t, 4) để trộn hai mảnh liên tiếp trong k thành một mảnh trong t ... Như vậy k và t được sử dụng với vai trò luân phiên: một dãy chứa các mảnh và một dãy dùng để trộn các cặp mảnh liên tiếp để được mảnh lớn hơn.

```

procedure MergeSort;
var
  t: TArray; {Dãy khoá phụ}
  len: Integer;
  Flag: Boolean; {Flag = True: trộn các mảnh trong k vào t; Flag = False: trộn các mảnh trong t vào k}

procedure Merge(var X, Y: TArray; a, b, c: Integer);{Trộn X[a..b] và X[b+1..c]}
var
  i, j, p: Integer;
begin
  {Chi số p chạy trong miền sắp xếp, i chạy theo mảnh thứ nhất, j chạy theo mảnh thứ hai}
  p := a; i := a; j := b + 1;
  while (i ≤ b) and (j ≤ c) then {Chừng nào cả hai mảnh đều chưa xét hết}
    begin
      if X[i] ≤ X[j] then {So sánh hai khoá nhỏ nhất trong hai mảnh mà chưa bị đưa vào miền sắp xếp}
        begin
          Y[p] := X[i]; i := i + 1; {Đưa x[i] vào miền sắp xếp và cho i chạy}
        end
      else
        begin
          Y[p] := X[j]; j := j + 1; {Đưa x[j] vào miền sắp xếp và cho j chạy}
        end;
      p := p + 1;
    end;
  if i ≤ b then Y[p..c] := X[i..b] {Mảnh 2 hết trước, Đưa phần cuối của mảnh 1 vào miền sắp xếp}
  else Y[p..c] := X[j..c]; {Mảnh 1 hết trước, Đưa phần cuối của mảnh 2 vào miền sắp xếp}
end;

procedure MergeByLength(var X, Y: TArray; len: Integer);
begin
  a := 1; b := len; c := 2 * len;
  while c ≤ n do {Trộn hai mảnh x[a..b] và x[b+1..c] đều có độ dài len}
    begin
      Merge(X, Y, a, b, c);
      a := a + 2 * len; b := b + 2 * len; c := c + 2 * len; {Dịch các chỉ số a, b, c về sau 2.len vị trí}
    end;
  if b < n then Merge(X, Y, a, b, n) {Còn lại hai mảnh mà mảnh thứ hai có độ dài ngắn hơn len}
  else
    if a ≤ n then Y[a..n] := X[a..n] {Còn lại một mảnh thì đưa thẳng mảnh đó sang miền y}
  end;

begin {Thuật toán sắp xếp trộn}
  Flag := True;
  len := 1;
  while len < n do
    begin
      if Flag then MergeByLength(k, t, len) else MergeByLength(t, k, len);
      len := len * 2;
      Flag := not Flag; {Đảo cờ để luôn phiên vai trò của k và t}
    end;
  if not Flag then k := t; {Nếu kết quả cuối cùng đang nằm trong t thì sao chép kết quả vào k}
end;

```

Về độ phức tạp của thuật toán, ta thấy rằng trong thủ tục Merge, phép toán tích cực là thao tác đưa một khoá vào miền sắp xếp. Mỗi lần gọi thủ tục MergeByLength, tất cả các khoá trong dãy khoá được chuyển hoàn toàn sang miền sắp xếp, nên độ phức tạp của thủ tục MergeByLength là  $O(n)$ . Thủ tục MergeSort có vòng lặp thực hiện không quá  $\lceil \lg n \rceil$  lần gọi MergeByLength bởi biến len sẽ được tăng theo cấp số nhân công bội 2. Từ đó suy ra độ phức tạp của MergeSort là  $O(n \lg n)$  bất chấp trạng thái dữ liệu vào.

Cùng là những thuật toán sắp xếp tổng quát với độ phức tạp trung bình như nhau, nhưng không giống như QuickSort hay HeapSort, MergeSort có tính  **ổn định**. Nhược điểm của MergeSort là nó phải dùng thêm một vùng nhớ để chứa dãy khoá phụ có kích thước bằng dãy khoá ban đầu.

Người ta còn có thể lợi dụng được trạng thái dữ liệu vào để khiến MergeSort chạy nhanh hơn: ngay từ đầu, ta không coi mỗi khoá của dãy khoá là một mảnh mà coi những đoạn đã được sắp trong dãy khoá là một mảnh. Bởi một dãy khoá bất kỳ có thể coi là gồm các mảnh đã sắp xếp nằm liên tiếp nhau. Khi đó người ta gọi phương pháp này là phương pháp  **trộn hai đường tự nhiên**.

Tổng quát hơn nữa, thay vì phép trộn hai mảnh, người ta có thể sử dụng phép trộn k mảnh, khi đó ta được thuật toán sắp xếp trộn k đường.

## 8.12. CÀI ĐẶT

Ta sẽ cài đặt tất cả các thuật toán sắp xếp nêu trên, với dữ liệu vào được đặt trong file văn bản SORT.INP chứa không nhiều hơn  $10^6$  khoá và giá trị mỗi khoá là số tự nhiên không quá  $10^6$ . Kết quả được ghi ra file văn bản SORT.OUT chứa dãy khoá được sắp, mỗi khoá trên một dòng.

| SORT.INP  | SORT.OUT |
|-----------|----------|
| 1 4 3 2 5 | 1        |
| 7 9 8     | 2        |
| 10 6      | 3        |
|           | 4        |
|           | 5        |
|           | 6        |
|           | 7        |
|           | 8        |
|           | 9        |
|           | 10       |

Chương trình có giao diện dưới dạng menu, mỗi chức năng tương ứng với một thuật toán sắp xếp. Tại mỗi thuật toán sắp xếp, ta thêm một vài lệnh đo thời gian thực tế của nó (chỉ đo thời gian thực hiện giải thuật, không tính thời gian nhập liệu và in kết quả).

Ở thuật toán Radix Exchange Sort, ta chọn hệ nhị phân. Ở thuật toán Straight Radix Sort, ta sử dụng hệ cơ số 256, khi đó nếu một giá trị số tự nhiên x biểu diễn bằng  $d + 1$  chữ số trong hệ 256:  $x = \overline{x_d \dots x_1 x_0}_{(256)}$  thì  $x_p = x \text{ div } 256^p \text{ mod } 256 = (x \text{ shr } (p \text{ shl } 3)) \text{ and } \$FF$  ( $1 \leq p \leq d$ ).

### P\_2\_08\_1.PAS \* Các thuật toán sắp xếp

```

{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)
program Sorting_Algorithms_Demonstration_Program;
uses crt, dos;
const
  InputFile = 'SORT.INP';
  OutputFile = 'SORT.OUT';
  max = 1000000;
  maxV = 1000000;
  BitCount = 64;
  nMenu = 13;
  SMenu: array[1..nMenu] of String =
  (

```

```
'D. Display Input',
'1. SelectionSort',
'2. BubbleSort',
'3. InsertionSort',
'4. InsertionSort with binary searching',
'5. ShellSort',
'6. QuickSort',
'7. HeapSort',
'8. Distribution Counting',
'9. Radix Exchange Sort',
'A. Straight Radix Sort',
'B. MergeSort',
'E. Exit'
);
type
  TArr = array[1..max] of Integer;
  TCount = array[0..maxV] of Integer;
var
  k, t: TArr;
  c: TCount;
  n, MinV, SupV: Integer;
  selected: Integer;
  StTime: Extended;

function GetcurrentTime: Extended;
var
  h, m, s, s100: Word;
begin
  GetTime(h, m, s, s100);
  GetcurrentTime := (h * 3600 + m * 60 + s) + s100 / 100;
end;

procedure Enter;
var
  f: Text;
begin
  Assign(f, InputFile); Reset(f);
  n := 0;
  MinV := High(Integer); SupV := 0;
  while not SeekEof(f) do
    begin
      Inc(n); Read(f, k[n]);
      if k[n] < MinV then MinV := k[n];
      if k[n] > SupV then SupV := k[n];
    end;
  Close(f);
  Inc(SupV);
  StTime := GetcurrentTime;
end;

procedure PrintInput;
var
  i: Integer;
begin
  Enter;
  for i := 1 to n do Write(k[i]:8);
  Write('Press any key to return to menu... ');
  ReadKey;
end;

procedure PrintResult;
var
  f: Text;
```

```

i: Integer;
ch: Char;
begin
  Writeln('Running Time = ', GetCurrentTime - StTime:1:4, ' (s)');
  Assign(f, OutputFile); Rewrite(f);
  for i := 1 to n do Writeln(f, k[i]);
  Close(f);
  Write('Press <P> to print Output, another key to return to menu... ');
  ch := ReadKey; Writeln(ch);
  if Upcase(ch) = 'P' then
    begin
      for i := 1 to n do Write(k[i]:8);
      Writeln;
      Write('Press any key to return to menu... ');
      ReadKey;
    end;
  end;

procedure Swap(var x, y: Integer);
var
  t: Integer;
begin
  t := x; x := y; y := t;
end;

{----- Sorting Algorithms -----}
{ SelectionSort }

procedure SelectionSort;
var
  i, j, jmin: Integer;
begin
  Enter;
  for i := 1 to n - 1 do
    begin
      jmin := i;
      for j := i + 1 to n do
        if k[j] < k[jmin] then jmin := j;
      if jmin <> i then Swap(k[i], k[jmin]);
    end;
  PrintResult;
end;

{ BubbleSort }

procedure BubbleSort;
var
  i, j: Integer;
begin
  Enter;
  for i := 2 to n do
    for j := n downto i do
      if k[j - 1] > k[j] then Swap(k[j - 1], k[j]);
  PrintResult;
end;

{ InsertionSort }

procedure InsertionSort;
var
  i, j, tmp: Integer;
begin
  Enter;

```

```

for i := 2 to n do
begin
  tmp := k[i]; j := i - 1;
  while (j > 0) and (tmp < k[j]) do
    begin
      k[j + 1] := k[j];
      Dec(j);
    end;
  k[j + 1] := tmp;
end;
PrintResult;
end;

{ InsertionSort with Binary searching }

procedure AdvancedInsertionSort;
var
  i, inf, sup, median, tmp: Integer;
begin
  Enter;
  for i := 2 to n do
    begin
      tmp := k[i];
      inf := 1; sup := i - 1;
      repeat
        median := (inf + sup) shr 1;
        if tmp < k[median] then sup := median - 1
        else inf := median + 1;
      until inf > sup;
      Move(k[inf], k[inf + 1], (i - inf) * SizeOf(k[1]));
      k[inf] := tmp;
    end;
  PrintResult;
end;

{ ShellSort }

procedure ShellSort;
var
  tmp: Integer;
  i, j, h: Integer;
begin
  Enter;
  h := n shr 1;
  while h <> 0 do
    begin
      for i := h + 1 to n do
        begin
          tmp := k[i]; j := i - h;
          while (j > 0) and (k[j] > tmp) do
            begin
              k[j + h] := k[j];
              j := j - h;
            end;
          k[j + h] := tmp;
        end;
      h := h shr 1;
    end;
  PrintResult;
end;

{ QuickSort }

```

```

procedure QuickSort;

procedure Partition(L, H: Integer);
var
  i, j: Integer;
  Pivot: Integer;
begin
  if L >= H then Exit;
  Pivot := k[L + Random(H - L + 1)];
  i := L; j := H;
  repeat
    while k[i] < Pivot do Inc(i);
    while k[j] > Pivot do Dec(j);
    if i <= j then
      begin
        if i < j then Swap(k[i], k[j]);
        Inc(i); Dec(j);
      end;
    until i > j;
  Partition(L, j); Partition(i, H);
end;

begin
  Enter;
  Partition(1, n);
  PrintResult;
end;

{ HeapSort }

procedure HeapSort;
var
  r, i: Integer;

procedure Adjust(root, endnode: Integer);
var
  key, c: Integer;
begin
  key := k[root];
  while root shr 1 <= endnode do
    begin
      c := root shr 1;
      if (c < endnode) and (k[c] < k[c + 1]) then Inc(c);
      if k[c] <= key then Break;
      k[root] := k[c]; root := c;
    end;
  k[root] := key;
end;

begin
  Enter;
  for r := n shr 1 downto 1 do Adjust(r, n);
  for i := n downto 2 do
    begin
      Swap(k[1], k[i]);
      Adjust(1, i - 1);
    end;
  PrintResult;
end;

{ Distribution Counting }

procedure DistributionCounting;

```

```

var
  i, V: Integer;
begin
  Enter;
  FillChar(c, SizeOf(c), 0);
  for i := 1 to n do Inc(c[k[i]]);
  for V := MinV + 1 to SupV - 1 do c[V] := c[V - 1] + c[V];
  for i := n downto 1 do
    begin
      V := k[i];
      t[c[V]] := k[i];
      Dec(c[V]);
    end;
  k := t;
  PrintResult;
end;

{ Radix Exchange Sort }

procedure RadixExchangeSort;
var
  MaskBit: array[0..BitCount - 1] of Integer;
  i, maxbit: Integer;

procedure Partition(L, H, BIndex: Integer);
var
  i, j, Mask: Integer;
begin
  if L >= H then Exit;
  i := L; j := H; Mask := MaskBit[BIndex];
  repeat
    while (i < j) and (k[i] and Mask = 0) do Inc(i);
    while (i < j) and (k[j] and Mask <> 0) do Dec(j);
    Swap(k[i], k[j]);
  until i = j;
  if k[j] and Mask = 0 then Inc(j);
  if BIndex > 0 then
    begin
      Partition(L, j - 1, BIndex - 1); Partition(j, H, BIndex - 1);
    end;
end;

begin
  Enter;
  maxbit := Trunc(Ln(SupV) / Ln(2));
  for i := 0 to maxbit do MaskBit[i] := 1 shl i;
  Partition(1, n, maxbit);
  PrintResult;
end;

{ Straight Radix Sort}

procedure StraightRadixSort;
const
  Radix = 256;
var
  p, maxDigit: Integer;
  Flag: Boolean;

  function GetDigit(key, p: Integer): Integer;
begin
  GetDigit := (key shr (p shl 3)) and $FF;
end;

```

```

procedure DCount(var x, y: TArr; p: Integer);
var
  c: array[0..Radix - 1] of Integer;
  i, d: Integer;
begin
  FillChar(c, SizeOf(c), 0);
  for i := 1 to n do
    begin
      d := GetDigit(x[i], p); Inc(c[d]);
    end;
  for d := 1 to Radix - 1 do c[d] := c[d - 1] + c[d];
  for i := n downto 1 do
    begin
      d := GetDigit(x[i], p);
      y[c[d]] := x[i];
      Dec(c[d]);
    end;
end;

begin
  Enter;
  MaxDigit := Trunc(Ln(SupV) / Ln(Radix));
  Flag := True;
  for p := 0 to MaxDigit do
    begin
      if Flag then DCount(k, t, p)
      else DCount(t, k, p);
      Flag := not Flag;
    end;
  if not Flag then k := t;
  PrintResult;
end;

{ MergeSort }

procedure MergeSort;
var
  Flag: Boolean;
  len: Integer;

procedure Merge(var Source, Dest: TArr; a, b, c: Integer);
var
  i, j, p: Integer;
begin
  p := a; i := a; j := b + 1;
  while (i <= b) and (j <= c) do
    begin
      if Source[i] <= Source[j] then
        begin
          Dest[p] := Source[i]; Inc(i);
        end
      else
        begin
          Dest[p] := Source[j]; Inc(j);
        end;
      Inc(p);
    end;
  if i <= b then
    Move(Source[i], Dest[p], (b - i + 1) * SizeOf(Source[1]))
  else
    Move(Source[j], Dest[p], (c - j + 1) * SizeOf(Source[1]));
end;

```

```

procedure MergeByLength(var Source, Dest: TArr; len: Integer);
var
  a, b, c: Integer;
begin
  a := 1; b := len; c := len shl 1;
  while c <= n do
    begin
      Merge(Source, Dest, a, b, c);
      a := a + len shl 1; b := b + len shl 1; c := c + len shl 1;
    end;
  if b < n then Merge(Source, Dest, a, b, n)
  else
    if a <= n then
      Move(Source[a], Dest[a], (n - a + 1) * SizeOf(Source[1]));
  end;

begin
  Enter;
  len := 1; Flag := True;
  FillChar(t, SizeOf(t), 0);
  while len < n do
    begin
      if Flag then MergeByLength(k, t, len)
      else MergeByLength(t, k, len);
      len := len shl 1;
      Flag := not Flag;
    end;
  if not Flag then k := t;
  PrintResult;
end;
{----- End of Sorting Algorithms -----}

function MenuSelect: Integer;
var
  i: Integer;
  ch: Char;
begin
  Clrscr;
  Writeln('Sorting Algorithms Demos; Input: SORT.INP; Output: SORT.OUT');
  for i := 1 to nMenu do Writeln(' ', SMenu[i]);
  Write('Enter your choice: ');
  ch := Upcase(ReadKey);
  Writeln(ch);
  for i := 1 to nMenu do
    if SMenu[i][1] = ch then
      begin
        MenuSelect := i;
        Exit;
      end;
  MenuSelect := 0;
end;

begin
  repeat
    selected := MenuSelect;
    if not (Selected in [1..nMenu]) then Continue;
    Writeln(SMenu[selected]);
    case selected of
      1 : PrintInput;
      2 : SelectionSort;
      3 : BubbleSort;
      4 : InsertionSort;
    end;
  until false;
end.

```

```

5 : AdvancedInsertionSort;
6 : ShellSort;
7 : QuickSort;
8 : HeapSort;
9 : DistributionCounting;
10: RadixExchangeSort;
11: StraightRadixSort;
12: MergeSort;
13: Halt;
end;
until False;
end.

```

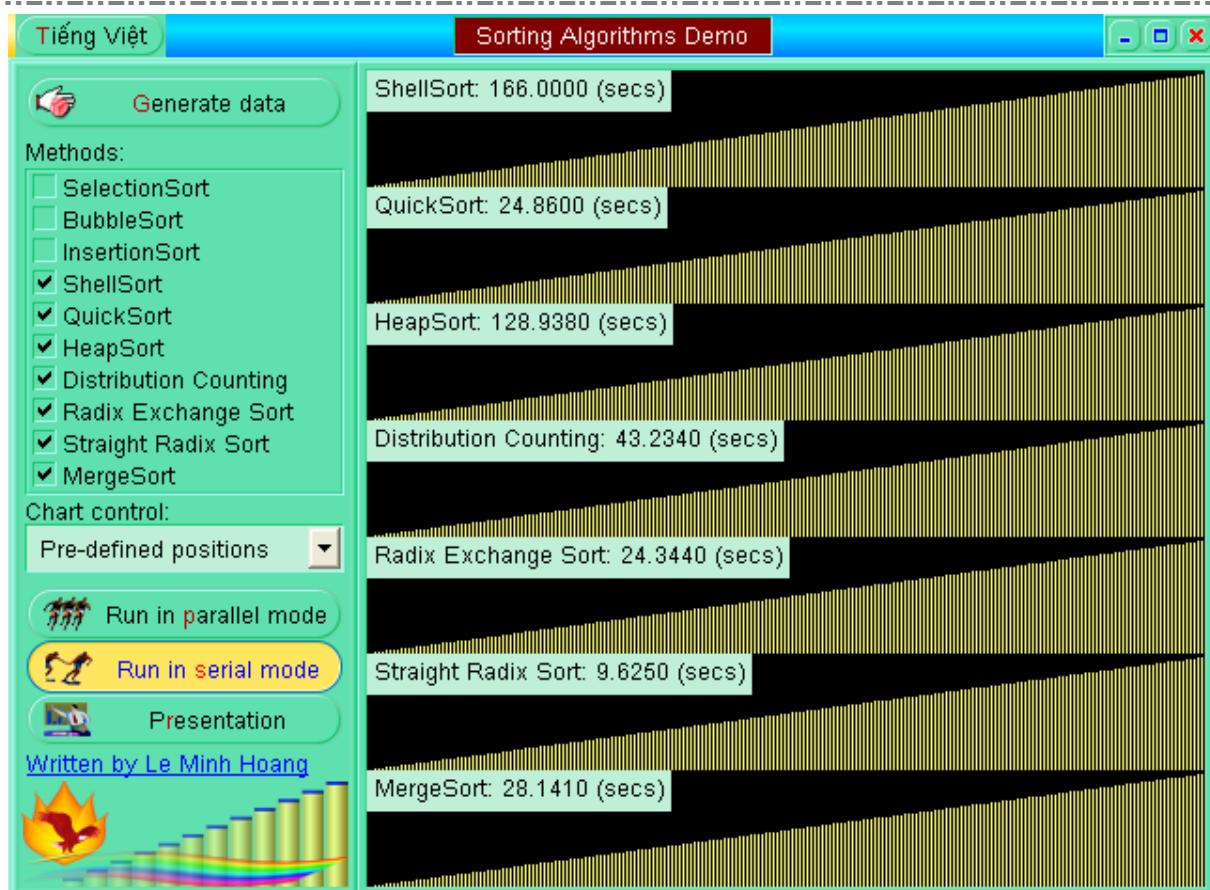
### 8.13. ĐÁNH GIÁ, NHẬN XÉT

Những con số về thời gian và tốc độ chương trình đo được là qua thử nghiệm trên một bộ dữ liệu cụ thể, với một máy tính cụ thể và một công cụ lập trình cụ thể. Với bộ dữ liệu khác, máy tính và công cụ lập trình khác, kết quả có thể khác. Tôi đã viết lại chương trình này trên Borland Delphi 7 để đưa vào một số cài tiến:

- ❖ Thiết kế dựa trên kiến trúc đa luồng (MultiThreads) cho phép chạy đồng thời hai hay nhiều thuật toán sắp xếp để so sánh tốc độ, bên cạnh đó vẫn có thể chạy tuần tự các thuật toán sắp xếp để đo thời gian thực hiện chính xác của chúng.
- ❖ Quá trình sắp xếp được hiển thị trực quan trên màn hình.
- ❖ Bỏ đi thuật toán sắp xếp kiểu chèn dạng nguyên thuỷ, chỉ giữ lại thuật toán sắp xếp kiểu chèn dùng tìm kiếm nhị phân
- ❖ Thuật toán ShellSort được viết lại, dùng các giá trị trong dãy Pratt: 1, 2, 3, 4, 6, 8, 9, 12, 16, ...,  $2^i 3^j$ , ... làm độ dài bước.

Thử nghiệm cả hai chương trình, một trên Free Pascal 1.0.10 và một trên Delphi 2005, nhìn chung tốc độ sắp xếp của chương trình viết trên Delphi nhanh hơn nhiều so với chương trình trên FPK, tuy nhiên khi so sánh tốc độ tương đối giữa các thuật toán vẫn có nhiều khác biệt giữa hai chương trình. Có một số thuật toán thực hiện nhanh bất ngờ so với dự đoán và cũng có một số thuật toán thực hiện chậm hơn hẳn so với đánh giá lý thuyết. Có thể có gắng giải thích qua hiệu ứng của bộ nhớ Cache và cách thức tối ưu mã lệnh, nhưng điều này hơi phức tạp và cũng không thực sự cần thiết. Ta chỉ rút ra kinh nghiệm rằng tốc độ thực thi của một thuật toán phụ thuộc rất nhiều vào phần cứng máy tính và chương trình dịch.

Hình 37 là giao diện của chương trình viết trên Delphi, bạn có thể tham khảo mã nguồn kèm theo. Có một điều phải lưu ý là để chương trình không bị ảnh hưởng bởi các phần mềm khác đang chạy, khi khởi động các threads, bàn phím, chuột và tất cả các phần mềm khác sẽ bị treo tạm thời đến khi các threads thực hiện xong. Vì vậy không nên chạy các thuật toán sắp xếp chậm với dữ liệu lớn vì sẽ không thể đợi đến khi các threads kết thúc và sẽ phải tắt máy khởi động lại.



**Hình 37: Máy Pentium 4, 3.2GHz, 2GB RAM tỏ ra chậm chạp khi sắp xếp  $10^8$  khoá  $\in [0..7.10^7]$  cho dù những thuật toán sắp xếp tốt nhất đã được áp dụng**

Cùng một mục đích sắp xếp như nhau, nhưng có nhiều phương pháp giải quyết khác nhau. Nếu chỉ dựa vào thời gian đo được trong một ví dụ cụ thể mà đánh giá thuật toán này tốt hơn thuật toán kia về mọi mặt là điều không nên. Việc chọn một thuật toán sắp xếp thích hợp cho phù hợp với từng yêu cầu, từng điều kiện cụ thể là kỹ năng của người lập trình.

Những thuật toán có độ phức tạp  $O(n^2)$  thì chỉ nên áp dụng trong chương trình có ít lần sắp xếp và với kích thước n nhỏ. Về tốc độ, BubbleSort luôn đứng bét, nhưng mã lệnh của nó lại hết sức đơn giản mà người mới học lập trình nào cũng có thể cài đặt được, tính ổn định của BubbleSort cũng rất đáng chú ý. Trong những thuật toán có độ phức tạp  $O(n^2)$ , InsertionSort tỏ ra nhanh hơn những phương pháp còn lại và cũng có tính ổn định, mã lệnh cũng tương đối đơn giản, dễ nhớ. SelectionSort thì không ổn định nhưng với n nhỏ, việc chọn ra m khoá nhỏ nhất có thể thực hiện dễ dàng chứ không cần phải sắp xếp lại toàn bộ như sắp xếp chèn.

Thuật toán đếm phân phối và thuật toán sắp xếp bằng cơ số nên được tận dụng trong trường hợp các khoá sắp xếp là số tự nhiên (hay là một kiểu dữ liệu có thể quy ra thành các số tự nhiên) bởi những thuật toán này có tốc độ rất cao. Thuật toán sắp xếp bằng cơ số cũng có thể sắp xếp dãy khoá có số thực hay số âm nhưng cần đưa vào một số sửa đổi nhỏ.

QuickSort, HeapSort, MergeSort và ShellSort là những thuật toán sắp xếp tổng quát, dãy khoá thuộc kiểu dữ liệu có thứ tự nào cũng có thể áp dụng được chứ không nhất thiết phải là các số.

QuickSort gặp nhược điểm trong trường hợp suy biến nhưng xác suất xảy ra trường hợp này rất nhỏ. HeapSort thì mã lệnh hơi phức tạp và khó nhớ, nhưng nếu cần chọn ra m khoá lớn nhất trong dãy khoá thì dùng HeapSort sẽ không phải sắp xếp lại toàn bộ dãy. MergeSort phải đòi hỏi thêm một không gian nhớ phụ, nên áp dụng nó trong trường hợp sắp xếp trên file. Còn ShellSort thì hơi khó trong việc đánh giá về thời gian thực thi, nó là sửa đổi của thuật toán sắp xếp chèn nhưng lại có tốc độ tương đối tốt, mã lệnh đơn giản và lượng bộ nhớ cần huy động rất ít. Tuy nhiên, những nhược điểm của bốn phương pháp này quá nhỏ so với ưu điểm chung của chúng là nhanh. Hơn nữa, chúng được đánh giá cao không chỉ vì tính tổng quát và tốc độ nhanh, mà còn là kết quả của những cách tiếp cận khoa học đối với bài toán sắp xếp.

Những thuật toán trên không chỉ đơn thuần là cho ta hiểu thêm về một cách sắp xếp mới, mà việc cài đặt chúng cũng cho chúng ta thêm nhiều kinh nghiệm: Kỹ thuật sử dụng số ngẫu nhiên, kỹ thuật “chia để trị”, kỹ thuật dùng các biến với vai trò luân phiên v.v... Vậy nên nắm vững nội dung của những thuật toán đó, mà cách thuộc tốt nhất chính là cài đặt chúng vài lần với các ràng buộc dữ liệu khác nhau (nếu có thể thử được trên hai ngôn ngữ lập trình thì rất tốt) và cũng đừng quên kỹ thuật sắp xếp bằng chỉ số.

## Bài tập

### Bài 1

Tìm hiểu các tài liệu khác để chứng minh rằng: Bất cứ thuật toán sắp xếp tổng quát nào dựa trên phép so sánh giá trị hai khoá đều có độ phức tạp tính toán trong trường hợp xấu nhất là  $\Omega(n \lg n)$ . (Sử dụng mô hình cây quyết định – Decision Tree Model).

### Bài 2

Viết thuật toán QuickSort không đệ quy

### Bài 3

Cho một danh sách thí sinh gồm n người, mỗi người cho biết tên và điểm thi, hãy chọn ra m người điểm cao nhất. Giải quyết bằng thuật toán có độ phức tạp O(n).

### Bài 4

Có 2 tính chất quan trọng của thuật toán sắp xếp: Stability và In place:

- ❖ Stability: Tính ổn định
- ❖ In place: Giải thuật không yêu cầu thêm không gian nhớ phụ, điều này cho phép sắp xếp một số lượng lớn các khoá mà không cần cấp phát thêm bộ nhớ. (Tuy vậy việc cấp phát thêm một lượng bộ nhớ  $\Theta(1)$  vẫn được cho phép)

Trong những thuật toán ta đã khảo sát, những thuật toán nào là stability, thuật toán nào là in place, thuật toán nào có cả hai tính chất trên.

### Bài 5

Cho một mảng a[1..n]

- Tìm giá trị xuất hiện nhiều hơn  $n/2$  lần trong a hoặc thông báo rằng không tồn tại giá trị như vậy. Tìm giải thuật với độ phức tạp O(n)

b) Cho số tự nhiên k, liệt kê tất cả các giá trị xuất hiện nhiều hơn n/k lần trong a. Tìm giải thuật với độ phức tạp O(n.k)

Cách giải:

a) Nếu một giá trị xuất hiện nhiều hơn n/2 lần trong a, giá trị đó phải là trung vị của dãy a. Ta sẽ tìm trung vị của dãy và duyệt lại dãy một lần nữa để xác nhận trung vị có xuất hiện nhiều hơn n/2 không.

b) Nếu một giá trị xuất hiện nhiều hơn n/k lần trong a, thì khi sắp xếp dãy a theo thứ tự không giảm, giá trị đó phải nằm ở một trong các vị trí  $\frac{i \times n}{k}$  ( $1 \leq i \leq k$ ). Áp dụng thuật toán thứ tự thông kê để tìm phần tử lớn thứ  $n/k, 2n/k, \dots, (k-1)n/k, n$ , và duyệt lại dãy để xác định giá trị của phần tử nào xuất hiện nhiều hơn n/k lần.

Bài 6

Thuật toán sắp xếp bằng cơ sở trực tiếp có ổn định không ? Tại sao ?

Bài 7

Cài đặt thuật toán sắp xếp trộn hai đường tự nhiên

Bài 8

Tìm hiểu phép trộn k đường và các phương pháp sắp xếp ngoài (trên tệp truy nhập tuần tự và tệp truy nhập ngẫu nhiên)

## §9. TÌM KIẾM (SEARCHING)

### 9.1. BÀI TOÁN TÌM KIẾM

Cùng với sắp xếp, tìm kiếm là một đòi hỏi rất thường xuyên trong các ứng dụng tin học. Bài toán tìm kiếm có thể phát biểu như sau:

Cho một dãy gồm n bản ghi  $r[1..n]$ . Mỗi bản ghi  $r[i]$  ( $1 \leq i \leq n$ ) tương ứng với một khoá  $k[i]$ . Hãy tìm bản ghi có giá trị khoá bằng X cho trước.

X được gọi là khoá tìm kiếm hay đối trị tìm kiếm (argument).

Công việc tìm kiếm sẽ hoàn thành nếu như có một trong hai tình huống sau xảy ra:

- ❖ Tìm được bản ghi có khoá tương ứng bằng X, lúc đó phép tìm kiếm thành công.
- ❖ Không tìm được bản ghi nào có khoá tìm kiếm bằng X cả, phép tìm kiếm thất bại.

Tương tự như sắp xếp, ta coi khoá của một bản ghi là đại diện cho bản ghi đó. Và trong một số thuật toán sẽ trình bày dưới đây, ta coi kiểu dữ liệu cho mỗi khoá cũng có tên gọi là TKey.

```
const
  n = ...; {Số khoá trong dãy khoá, có thể khai dưới dạng biến số nguyên để tùy biến hơn}
type
  TKey = ...; {Kiểu dữ liệu một khoá}
  TArray = array[1..n] of TKey;
var
  k: TArray; {Dãy khoá}
```

### 9.2. TÌM KIẾM TUẦN TỤ (SEQUENTIAL SEARCH)

Tìm kiếm tuần tự là một kỹ thuật tìm kiếm đơn giản. Nội dung của nó như sau: Bắt đầu từ bản ghi đầu tiên, lần lượt so sánh khoá tìm kiếm với khoá tương ứng của các bản ghi trong danh sách, cho tới khi tìm thấy bản ghi mong muốn hoặc đã duyệt hết danh sách mà chưa thấy

```
{Tìm kiếm tuần tự trên dãy khoá k[1..n]; hàm này thử tìm xem trong dãy có khoá nào = X không, nếu thấy nó trả về chỉ số
của khoá ấy, nếu không thấy nó trả về 0. Có sử dụng một khoá phụ k[n+1] được gán giá trị = X}
function SequentialSearch(X: TKey): Integer;
var
  i: Integer;
begin
  i := 1;
  while (i <= n) and (k[i] ≠ X) do i := i + 1;
  if i = n + 1 then SequentialSearch := 0
  else SequentialSearch := i;
end;
```

Để thấy rằng độ phức tạp của thuật toán tìm kiếm tuần tự trong trường hợp tốt nhất là  $O(1)$ , trong trường hợp xấu nhất là  $O(n)$  và trong trường hợp trung bình là  $\Theta(n)$ .

### 9.3. TÌM KIẾM NHỊ PHÂN (BINARY SEARCH)

Phép tìm kiếm nhị phân có thể áp dụng trên dãy khoá đã có thứ tự:  $k[1] \leq k[2] \leq \dots \leq k[n]$ .

Giả sử ta cần tìm trong đoạn  $k[\inf..\sup]$  với khoá tìm kiếm là X, trước hết ta xét khoá nằm giữa dãy  $k[\text{median}]$  với  $\text{median} = (\inf + \sup) \text{ div } 2$ ;