CHƯƠNG 6. Sorting and Searching

NỘI DUNG:

6.1. Đặt vấn đề

6.2. Các thuật toán sắp xếp đơn giản

6.3. Thuật toán Quick-Sort

6.4. Thuật toán Merge-Sort

6.5. Thuật toán Heap-Sort

6.6. Thuật toán Shell-Sort

6.7. Thuật toán Radix-Sort

6.8. Tìm kiếm tuyến tính

6.9. Tìm kiếm nhị phân

6.10. Case Study

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.1. Giới thiệu bài toán

Một trong những vấn đề quan trọng bậc nhất của khoa học máy tính là bài toán tìm kiếm (Searching Problem). Có thể nói, hầu hết các hoạt động của người dùng hoặc các ứng dụng tin học có thể triển khai được đều liên quan đến tìm kiếm. Bài toán tìm kiếm có thể được phát biểu như sau:

Bài toán tìm kiếm: Cho dãy gồm n đối tượng r_1, r_2, .., r_n. Mỗi đối tượng r_i được tương ứng với một khóa k_i ($1 \le i \le n$). Nhiệm vụ của tìm kiếm là xây dựng thuật toán tìm đối tượng có giá trị khóa là X cho trước. X còn được gọi là khóa tìm kiếm hay tham biến tìm kiếm (arrgument).

Công việc tìm kiếm bao giờ cũng hoàn thành bởi một trong hai tình huống:

- Nếu tìm thấy đối tượng có khóa X trong tập các đối tượng thì ta nói phép tìm kiếm thành công (successful).
- Nếu không tìm thấy đối tượng có khóa X trong tập các đối tượng thì ta nói phép tìm kiếm không thành công (unsuccessful).

Trong tìm kiếm, ta luôn hiểu tìm kiếm các đối tượng tổng quát (các bản ghi). Để đơn giản, chúng ta xem mỗi đối tượng như một số trong các thuật toán trình được trình bày tiếp theo.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Sắp xếp là phương pháp bố trí lại các đối tượng theo một trật tự nào đó. Ví dụ bố trí theo thứ tự tăng dần hoặc giảm dần đối với dãy số, bố trị theo thứ tự từ điển đối với các xâu ký tự. Mục tiêu của sắp xếp là để lưu trữ và tìm kiếm đối tượng (thông tin) để đạt hiệu quả cao trong tìm kiếm. Có thể nói, sắp xếp là sân sau quả quá trình tìm kiếm. Muốn tìm kiếm và cung cấp thông tin nhanh thì ta cần phải sắp xếp thông tin sao cho hợp lý. Bài toán sắp xếp có thể được phát biểu như sau:

Bài toán sắp xếp: Cho dãy gồm n đối tượng r_1, r_2, .., r_n. Mỗi đối tượng r_i được tương ứng với một khóa k_i ($1 \leq i \leq n$). Nhiệm vụ của sắp xếp là xây dựng thuật toán bố trí các đối tượng theo một trật tự nào đó của các giá trị khóa.

Trong các mục tiếp theo, chúng ta xem tập các đối tượng cần sắp xếp là tập các số. Việc mở rộng các số cho các bản ghi tổng quát cũng được thực hiện tương tự bằng cách thay đổi các kiểu dữ liệu tương ứng. Cũng giống như tìm kiếm, việc làm này không làm mất đi bản chất của thuật toán.

Created using
easyPDF Printer
Click here to purchase a license to remove this image

6.2. Các thuật toán sắp xếp đơn giản

Các thuật toán sắp xếp đơn giản được trình bày ở đây bao gồm:

- Thuật toán sắp xếp kiểu lựa chọn (Selection Sort).
- Thuật toán sắp xếp kiểu chèn (Insertion Sort).
- Thuật toán sắp xếp kiểu sủi bọt (Bubble Sort).

6.2.1. Thuật toán Selection-Sort

Thuật toán sắp xếp đơn giản nhất được đề cập đến là thuật toán sắp xếp kiểu chọn. Thuật toán thực hiện sắp xếp dãy các đối tượng bằng cách lặp lại việc tìm kiếm phần tử có giá trị nhỏ nhất từ thành phần chưa được sắp xếp trong mảng và đặt nó vào vị trí đầu tiên của dãy. Trên dãy các đối tượng ban đầu, thuật toán luôn duy trì hai dãy con:

- Dãy con đã được sắp xếp: là các phần tử bên trái của dãy.
- Dãy con chưa được sắp xếp là các phần tử bên phải của dãy.

Quá trình lặp sẽ kết thúc khi dãy con chưa được sắp xếp chỉ còn lại đúng một phần tử.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Thuật toán Selection-Sort:

Input:

- Dãy các đối tượng (các số) : Arr[0], Arr[1],..,Arr[n-1].
- Số lượng các đối tượng cần sắp xếp: n.

Output:

- Dãy các đối tượng đã được sắp xếp (các số) : Arr[0], Arr[1],..,Arr[n-1].

Formats: Selection-Sort(Arr, n);

Actions:

```
for (i =0; i<n-1; i++) { //duyệt các phần tử i=0,1,.., n-1
    min_idx = i;  //gọi min_idx là vị trí của phần tử nhỏ nhất trong dãy con
    for ( j = i +1; j<n; j++ ) { //duyệt từ phần tử tiếp theo j=i+1,..,n.
        if (Arr[i] > Arr[j] ) // nếu Arr[i] không phải nhỏ nhất trong dãy con
            min_idx = j; //ghi nhận đây mới là vị trí phần tử nhỏ nhất.
    }
    //đặt phần tử nhỏ nhất vào vị trí đầu tiên của dãy con chưa được sắp
    Temp = Arr[i] ; Arr[i] = Arr[min_idx]; Arr[min_idx] = temp;
```

End.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm thuật toán: Arr[] = { 9, 7, 12, 8, 6, 5 }, n = 6.

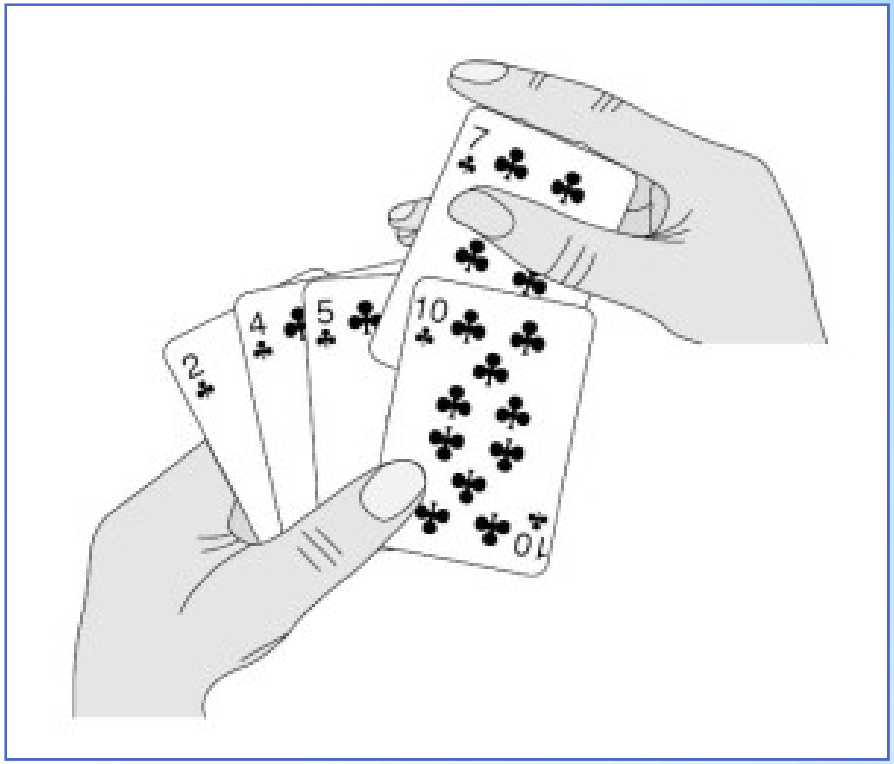
Bước	min-idx	Dãy số Arr[] =?
i = 0	min-idx=5	Arr[] = {**5**, 7, 12, 8, 6, 9}
i = 1	min-idx=4	Arr[] = {**5**, **6**, 12, 8, 7, 9}
i = 2	min-idx=4	Arr[] = {**5**, **6**, **7**, 8, 12, 9}
i = 3	min-idx=3	Arr[] = {**5**, **6**, **7**, **8**, 12, 9}
i = 4	min-idx=5	Arr[] = {**5**, **6**, **7**, **8**, **9**, 12}

Độ phức tạp thuật toán: $O(n^2)$, với n là số lượng phần tử của dãy.

Created using
easyPDF Printer
Click here to purchase a license to remove this image

6.2.2. Thuật toán Insertion-Sort

Thuật toán sắp xếp kiểu chèn được thực hiện đơn giản theo cách của người chơi bài thông thường. Phương pháp được thực hiện như sau:



- Lấy phần tử đầu tiên Arr[0] (quân bài đầu tiên) như vậy ta có dãy một phần tử được sắp.
- Lấy phần tiếp theo (quân bài tiếp theo) Arr[1] và tìm vị trí thích hợp chèn Arr[1] vào dãy Arr[0] để có dãy hai phần tử đã được sắp.
- Tổng quát, tại bước thứ i ta lấy phần tử thứ i và chèn vào dãy Arr[0],..,Arr[i-1] đã được sắp trước đó để nhận được dãy i phần tử được sắp
- Quá trình sắp xếp sẽ kết thúc khi i = n.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Thuật toán Insertion-Sort:

Input:

- Dãy các đối tượng (các số) : Arr[0], Arr[1],..,Arr[n-1].
- Số lượng các đối tượng cần sắp xếp: n.

Output:

- Dãy các đối tượng đã được sắp xếp (các số) : Arr[0], Arr[1],..,Arr[n-1].

Formats: Insertion-Sort(Arr, n);

Actions:

```
for (i = 1; i < n; i++) { //lặp i=1, 2,..,n.
        key = Arr[i]; //key là phần tử cần chèn váo dãy Arr[0],.., Arr[i-1]
        j = i-1;
        while (j >= 0 && Arr[j] > key) { //Duyệt lùi từ vị trí j=i-1
                Arr[j+1] = Arr[j]; //dịch chuyển Arr[j] lên vị trí Arr[j+1]
                j = j-1;
        }
        Arr[j+1] = key; // vị trí thích hợp của key trong dãy là Arr[j+1]
}
```

End.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm thuật toán: Arr[] = { 9, 7, 12, 8, 6, 5 }, n = 6.

Bước	Arr[j+1]=key	Dãy số Arr[] =?
i = 1	Arr[0] =9	Arr[] = {**7, 9,** 12, 8, 6, 5}
i = 2	Arr[2] =12	Arr[] = {**7, 9, 12**, 8, 6, 5}
i = 3	Arr[1] =8	Arr[] = {**7, 8, 9, 12**, 6, 5}
i = 4	Arr[0] =6	Arr[] = {**6, 7, 8, 9, 12**, 5}
i = 5	Arr[0] =5	Arr[] = {**5, 6, 7, 8, 9, 12**}

Độ phức tạp thuật toán: $O(n^2)$, với n là số lượng phần tử của dãy. Trong trường hợp tốt nhất dãy đã được sắp độ phức tạp là O(n).

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.2.3. Thuật toán Bubble-Sort

Thuật toán sắp xếp kiểu sủi bọt được thực hiện đơn giản bằng cách tráo đổi hai phần từ liền kề nhau nếu chúng chưa được sắp xếp. Thuật toán được thực hiện như sau:

Thuật toán Bubble-Sort:

Input:

- Dãy các đối tượng (các số) : Arr[0], Arr[1],..,Arr[n-1].
- Số lượng các đối tượng cần sắp xếp: n.

Output:

- Dãy các đối tượng đã được sắp xếp (các số) : Arr[0], Arr[1],..,Arr[n-1].

Formats: Insertion-Sort(Arr, n);

Actions:

```
for (i = 0; i < n; i++) { ///lặp i=0, 1, 2,..,n.
        for (j=0; j<n-i-1; j++ ) {///lặp j =0, 1,.., n-i-1
            if (Arr[j] > Arr[j+1] ) { //nếu Arr[j]>Arr[j+1] thì đổi chỗ
                temp = Arr[j];
                Arr[j] = Arr[j+1];
                Arr[j+1] = temp;
            }
        }
}
```

End.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm thuật toán: Arr[] = { 9, 7, 12, 8, 6, 5 }, n = 6.

Bước	Dãy số Arr[] =?
i = 0	Arr[] = {7, 9, 8, 6, 5, **12**}
i = 1	Arr[] = {7, 8, 6, 5, **9, 12**}
i = 2	Arr[] = {7, 6, 5, **8, 9, 12**}
i = 3	Arr[] = {6, 5, **7, 8, 9, 12**}
i = 4	Arr[] = {**5, 6, 7, 8, 9, 12**}
i = 5	Arr[] = {**5, 6, 7, 8, 9, 12**}

Độ phức tạp thuật toán: $O(n^2)$, với n là số lượng phần tử của dãy.

Created using
easyPDF Printer
Click here to purchase a license to remove this image

6. 3. Thuật toán Quick-Sort

Thuật toán sắp xếp Quick-Sort được thực hiện theo mô hình chia để trị (Devide and Conquer). Thuật toán được thực hiện xung quang một phần tử gọi là chốt (key). Mỗi cách lựa chọn vị trí phần tử chốt trong dãy sẽ cho ta một phiên bản khác nhau của thuật toán. Các phiên bản (version) của thuật toán Quick-Sort thông dụng là:

- Luôn lựa chọn phần tử đầu tiên trong dãy làm chốt.
- Luôn lựa chọn phần tử cuối cùng trong dãy làm chốt.
- Luôn lựa chọn phần tử ở giữa dãy làm chốt.
- Lựa chọn phần tử ngẫu nhiên trong dãy làm chốt.

Mấu chốt của thuật toán Quick-Sort là làm thế nào ta xây dựng được một thủ tục phân đoạn (Partition). Thủ tục Partition có hai nhiệm vụ chính:

- Định vị chính xác vị trí của chốt trong dãy nếu được sắp xếp;
- Chia dãy ban đầu thành hai dãy con: dãy con ở phía trước phần tử chốt bao gồm các phần tử nhỏ hơn hoặc bằng chốt, dãy ở phía sau chốt có giá trị lớn hơn chốt.

Có nhiều thuật toán khác nhay để xây dựng thủ tục Partition, dưới đây là một phương pháp xây dựng thủ tục Partion với khóa chốt là phần tử cuối cùng của dãy.

Created using easyPDF Printer
Click here to purchase a license to remove this image

6. 3. 1. Thuật toán Partition

Thuật toán Partition:

Input :

- Dãy Arr[] bắt đầu tại vị trí l và kết thúc tại h.
- Cận dưới của dãy con: l
- Cận trên của dãy con: h

Output:

- Vị trí chính xác của Arr[h] nếu dãy Arr[] được sắp xếp.

Formats: Partition(Arr, l, h);

Actions:

```
x = Arr[h];    // Chọn x là chốt của dãy Arr[l],Arr[l+1],.., Arr[h]
i = (l - 1);  // i là chỉ số của các phần tử nhỏ hơn chốt x.
for ( j = l;  j <= h- 1;  j++) { //duyệt các chỉ số j=l, l+1, …, h-1.
        if (Arr[j] <= x){ //nếu Arr[j] nhỏ hơn hoặc bằng chốt x.
                i++; //tăng vị trí các phần tử nhỏ hơn chốt
                swap(&Arr[i], &Arr[j]);  // đổi chỗ Arr[i] cho Arr[j].
        }
}
swap(&Arr[i + 1], &Arr[h]); // đổi chỗ Arr[i+1] cho Arr[h].
return (i + 1); //Vị trí i+1 chính là vị trí chốt nếu được sắp xếp
```

End.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm thuật toán Partition: p = Partition(Arr, 0, 9);

Arr[] = {10, 27, 15, 29, 21, 11, 14, 18, 12, 17};

l=0, h = 9. Khóa x = Arr[h] = 17. Vị trí bắt đầu i= l - 1 = -1.

Bước j=?	(Arr[j]<=x) ?	Đổi chỗ Arr[i] cho Arr[j]
j = 0 : i=0	(10<=17): Yes	Arr[] = {10, 27, 15, 29, 21, 11, 14, 18, 12, 17}
j = 1: i=0	(27<=17):No	Arr[] = {10, 27, 15, 29, 21, 11, 14, 18, 12, 17}
j = 2:i=1	(15<=17):Yes	Arr[] = {10, 15, 27, 29, 21, 11, 14, 18, 12, 17}
j = 3:i=1	(29<=17):No	Arr[] = {10, 15, 27, 29, 21, 11, 14, 18, 12, 17}
j = 4:i=1	(21<=17):No	Arr[] = {10, 15, 27, 29, 21, 11, 14, 18, 12, 17}
j = 5:i=2	(11<=17):Yes	Arr[] = {10, 15, 11, 29, 21, 27, 14, 18, 12, 17}
j = 6:i=3	(14<=17):Yes	Arr[] = {10, 15, 11, 14, 21, 27, 29, 18, 12, 17}
j = 7:i=3	(18<=17):No	Arr[] = {10, 15, 11, 14, 21, 27, 29, 18, 12, 17}
j = 8:i=4	(12<=17)Yes	Arr[] = {10, 15, 11, 14, 12, 27, 29, 18, 21, 17}
i+1 =5	Arr[] = {10, 15, 11, 14, 12} (17) { 29, 18, 21, 27}	
Kết luận p = i+1 = 5 là vị trí của khóa x =17 trong dãy Arr[] nếu được sắp xếp		

Created using easyPDF Printer
Click here to purchase a license to remove this image

6. 3. 2. Thuật toán Quick-Sort

Thuật toán Partition:

Input :

- Dãy Arr[] gồm n phần tử.
- Cận dưới của dãy: l.
- Cận trên của dãy : h

Output:

- Dãy Arr[] được sắp xếp.

Formats: Quick-Sort(Arr, l, h);

Actions:

```
if( l<h) { // Nếu cận dưới còn nhỏ hơn cận trên
        p = Partition(Arr, l, h); //thực hiện Partition() chốt h.
        Quick-Sort(Arr, l, p-1); //Thực hiện Quick-Sort nửa bên trái.
        Quick-Sort(Arr, p+1, h);//Thực hiện Quick-Sort nửa bên phải
}
```

End.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Độ phức tạp thuật toán:

- Trường hợp xấu nhất : $O(n^2)$.
- Trường hợp tốt nhất : $O(n \log(n))$.

Kiểm nghiệm thuật toán Quick-Sort: Quick-Sort(Arr, 0, 9);

Arr[] = {10, 27, 15, 29, 21, 11, 14, 18, 12, 17};

p = Partition(Arr,l,h)	Giá trị Arr[]=?
p=5:l=0, h=9	{10,15,11,14,12}, **(17)**,{29,18, 21, 27}
P=2:l=0, h=4	{10,11},**(12)**, {14,15}, (17),{29,18, 21, 27}
P=1:l=0, h=1	{10,**11**},(12), {14,15}, (17),{29,18, 21, 27}
P=4: l=3, h=4	{10,11},(12), {14,**15**}, (17),{29,18, 21, 27}
P=8: l=6, h=9	{10,11},(12), {14,15}, (17),{18,21},(**27**),{29}
P=7:l=6, h=7	{10,11},(12), {14,15}, (17),{18,**21**},(27),{29}
Kết luận dãy được sắp Arr[] ={ 10, 11, 12, 14, 15, 17, 18, 21, 27, 29}	

Created using easyPDF Printer
Click here to purchase a license to remove this image

6. 4. Thuật toán Merge-Sort

Giống như Quick-Sort, Merge-Sort cũng được xây dựng theo mô hình chia để trị (Devide and Conquer). Thuật toán chia dãy cần sắp xếp thành hai nửa. Sau đó gọi đệ qui lại cho mỗi nửa và hợp nhất lại các đoạn đã được sắp xếp. Thuật toán được tiến hành theo 4 bước dưới đây:

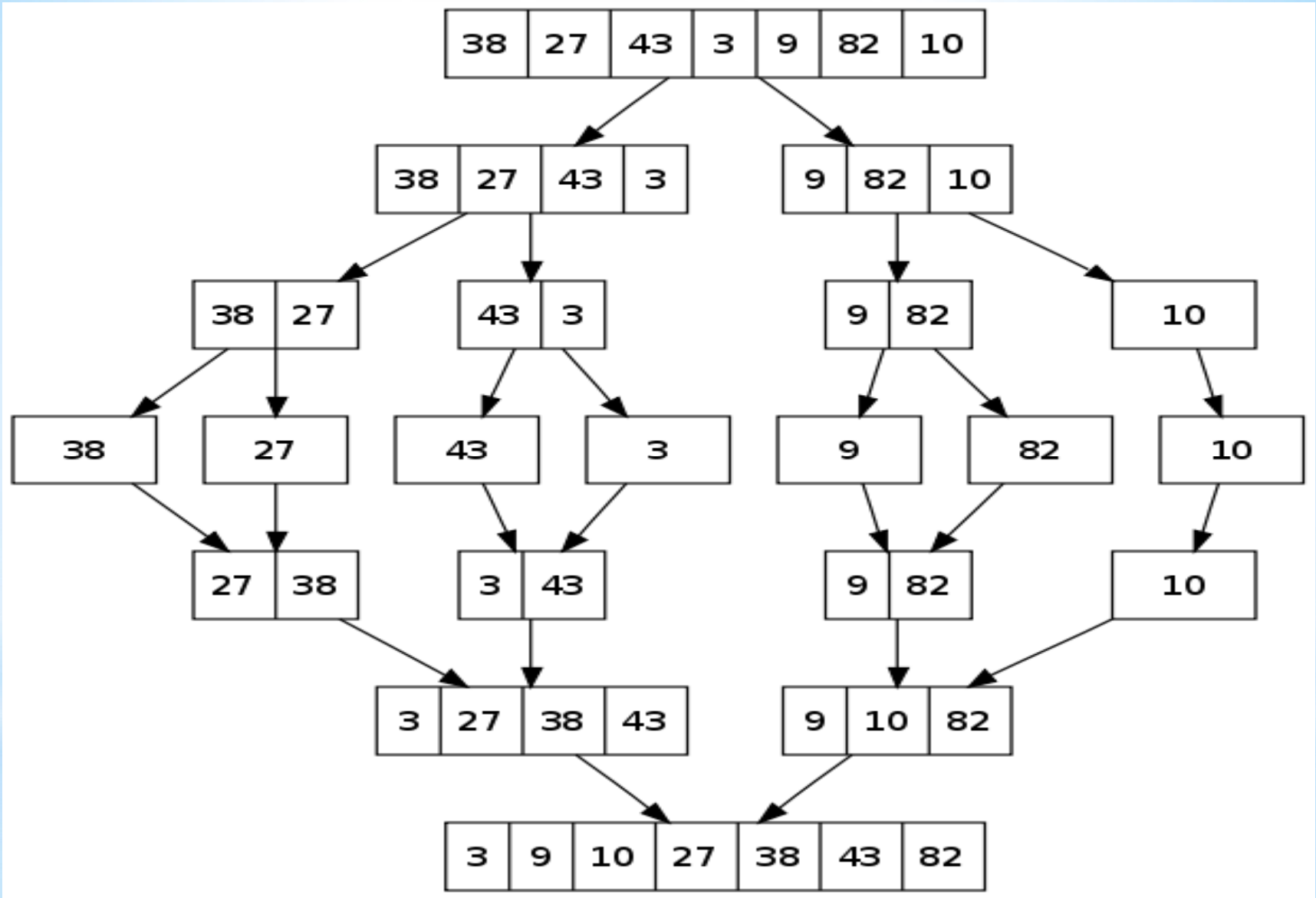
- Tìm điểm giữa của dãy và chi dãy thành hai nửa.
- Thực hiện Merge-Sort cho nửa thứ nhất.
- Thực hiện Merge-Sort cho nửa thứ hai.
- Hợp nhất hai đoạn đã được sắp xếp.

Mấu chốt của thuật toán Merge-Sort là làm thế nào ta xây dựng được một thủ tục hợp nhất (Merge). Thủ tục Merge thực hiện hòa nhập hai dãy đã được sắp xếp để tạo thành một dãy cũng được sắp xếp. Bài toán có thể được phát biểu như sau:

Bài toán hợp nhất Merge: Cho hai nửa của một dãy Arr[1,..,m] và A[m+1,..,r] đã được sắp xếp. Nhiệm vụ của ta là hợp nhất hai nửa của dãy Arr[1,..,m] và Arr[m+1,..,r] để trở thành một dãy Arr[1, 2,..,r] cũng được sắp xếp.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Quá trình thực hiện thuật toán Merge



Created using
easyPDF Printer
Click here to purchase a license to remove this image

6. 4. 1. Thuật toán Merge

```
void Merge( int Arr[],  int l,  int m,  int r){  int i, j, k, n1 = m - l + 1; n2 =  r - m;
        int L[n1], R[n2]; //tạo lập hai mảng phụ
        // Copy dữ liệu vào L[] và R[]
        for(i = 0; i < n1; i++)  L[i] = Arr[l + i];
        for(j = 0; j < n2; j++)  R[j] = Arr[m + 1+ j];
        // hợp nhất các mảng phụ và trả lại vào Arr[]
        i = 0; j = 0; k = l;
        while (i < n1 && j < n2){
                if (L[i] <= R[j]){ Arr[k] = L[i]; i++; }
                else { Arr[k] = R[j];  j++; }
                k++;
        }
         while (i < n1)  {
                Arr[k] = L[i];  i++; k++;
        }
        while (j < n2) {
                arr[k] = R[j];  j++; k++;
        }
}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm thuật toán Merge:

Input : Arr[] = { (19, 17), (**20, 22, 24**), (**15, 18, 23, 25**), (35, 28, 13)}
l = 2, m = 4, r =8.

Output : Arr[] = { (19, 17), (**15, 18, 20, 22, 23, 24, 25**), (35, 28, 13)}

Tính toán:

n1 = m-l+1 = 3; n2 = r-m= 5. //dòng thứ nhất của thuật toán
L = {20, 22, 24}. //vòng for thứ nhất
R = {15, 18, 23, 25}. //vòng for thứ hai

i=? j=?, k=?	(L[i]<=R[j])?	Arr[k] =?
i = 0; j=0, k=2	(20<=15):No	Arr[2] = 15;
i = 0; j=1, k=3	(20<=18):No	Arr[3] = 18;
i = 0; j=2, k=4	(20<=23):Yes	Arr[4] = 20;
i = 1; j=2, k=5	(22<=23):Yes	Arr[5] = 22;
i = 2; j=2, k=6	(24<=23):No	Arr[6] = 23;
i = 2; j=3, k=7	(24<=25):Yes	Arr[7] = 24;
((i=3)<3): No; j=3; k=7		
Kết quả: Arr[] = {(19,17),(**15,18,20, 22, 23, 24, 25**), (35, 28, 13)}		

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.4.2. Thuật toán Merge-Sort:

Input :

- Dãy số : Arr[];
- Cận dưới: l;
- Cận trên m;

Output:

- Dãy số Arr[] được sắp theo thứ tự tăng dần.

Formats: Merge-Sort(Arr, l, r);

Actions:

```
if ( l< r ) {
        m = (l + r -1) / 2; //phép chia Arr[] thành hai nửa
        Merge-Sort(Arr, l, m);//trị nửa thứ nhất
        Merge-Sort(Arr, m+1, r); //trị nửa thứ hai
        Merge(Arr, l, m, r); //hợp nhất hai nửa đã sắp xếp
}
```

End.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Độ phức tạp thuật toán: O(nlog n).

Kiểm nghiệm thuật toán Merge-Sort: Merge-Sort(Arr,0,7)

Input :

Arr[] = {38, 27, 43, 3, 9, 82, 10}; n = 7;

Bước	Kết quả Arr[]=?
1	Arr[] = { 27, 38, 43, 3, 9, 82, 10}
2	Arr[] = { 27, 38, 3, 43, 9, 82, 10}
3	Arr[] = { 3, 27, 38, 43, 9, 82, 10}
4	Arr[] = {3, 27, 38, 43, 9, 82, 10}
5	Arr[] = { 3, 27, 38, 43, 9, 10, 82}
6	Arr[]= { 3, 9, 10, 27, 38, 43, 82}

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.5. Thuật toán Heap-Sort

6.5.1. Giới thiệu thuật toán

Thuật toán Heap-Sort được thực hiện dựa trên cấu trúc dữ liệu Heap. Nếu ta muốn sắp xếp theo thứ tự tăng dần ta sử dụng cấu trúc Max Heap, ngược lại ta sử dụng cấu trúc Min-Heap. Vì Heap là một cây nhị phân đầy đủ nên việc biểu diễn Heap một cách hiệu quả có thể thực hiện được bằng mảng.

Tư tưởng của Heap Sort giống như Selection Sort, chọn phần tử lớn nhất trong dãy đặt vào vị trí cuối cùng, sau đó lặp lại quá trình này cho các phần tử còn lại. Tuy nhiên, điểm khác biệt ở đây là phần tử lớn nhất của Heap luôn là phần tử đầu tiên trên Heap. Chính vì lý do này, độ phức tạp thuật toán chỉ còn là O(nlog(n)).

Thuật toán được thực hiện thông qua các bước chính như sau:

1) Xây dựng Max Heap từ dữ liệu vào. Ví dụ với dãy A[] = {9, 7, 12, 8, 6, 5} thì Max Heap được xây dựng là A[] = {12, 8, 9, 7, 6, 5}.
2) Bắt đầu tại vị trí đầu tiên là phần tử lớn nhất của dãy. Thay thế, phần tử này cho phần tử cuối cùng ta nhận được dãy A[] = {5, 8, 9, 7, 6, 12}.
3) Xây dựng lại Max Heap cho n-1 phần tử đầu tiên của dãy và lặp lại quá trình này cho đến khi Heap chỉ còn lại 1 phần tử.

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.5.2. Thuật toán xây dựng Heap-Sort

```
void max_heapify(int *A, int i, int n){  int j, temp; temp = A[i]; j = 2*i;
        while (j <= n){
                if (j < n && A[j+1] > A[j])   j = j+1;
                if (temp > A[j])       break;
                else if (temp <= A[j]){ A[j/2] = A[j]; j = 2*j; }
                }
        }
        A[j/2] = temp;
}
void build_maxheap(int *A, int n){
        for( int i = n/2; i >= 1; i--)     max_heapify(A, i, n);
 }
void heapsort(int *A, int n){     int i, temp;
        for (i = n; i >= 2; i--){
                temp = A[i];  A[i] = A[1];  A[1] = temp; //Luôn đổi chỗ cho A[1]
                max_heapify(A, 1, i - 1); //tạo Max-Heap cho i-1 số còn lại
        }
}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.5.3. Kiểm nghiệm thuật toán

Giả sử ta cần sắp xếp dãy số A[] = { 9, 7, 12, 8, 6, 5, 13, 3, 18, 4}, n=10. Khi đó, kết quả thực hiện của thuật toán theo từng bước dưới đây:

Bước	Dãy số A[]
1	A[] = {18, 9, 13, 8, 6, 5, 12, 3, 7, 4}
2	A[] = { 13, 9, 12, 8, 6, 5, 4, 3, 7, 18}
3	A[] = {12, 9, 7, 8, 6, 5, 4, 3, 13, 18}
4	A[] = {9, 8, 7, 3, 6, 5, 4, 12, 13, 18}
5	A[] = {8, 6, 7, 3, 4, 5, 9, 12, 13, 18}
6	A[] = {7, 6, 5, 3, 4, 8, 9, 12, 13, 18}
7	A[] = {6, 4, 5, 3, 7, 8, 9, 12, 13, 18}
8	A[] = {5, 4, 3, 6, 7, 8, 9, 12, 13, 18}
9	A[] = {4, 3, 5, 6, 7, 8, 9, 12, 13, 18}
10	A[] = {3, 4, 5, 6, 7, 8, 9, 12, 13, 18}

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.6. Tìm kiếm (Searching)

Một trong những vấn đề quan trọng bậc nhất của khoa học máy tính là bài toán tìm kiếm (Searching Problem). Có thể nói, hầu hết các hoạt động của người dùng hoặc các ứng dụng tin học có thể triển khai được đều liên quan đến tìm kiếm. Bài toán tìm kiếm có thể được phát biểu như sau:

Bài toán tìm kiếm: Cho dãy gồm n đối tượng r_1, r_2, .., r_n. Mỗi đối tượng r_i được tương ứng với một khóa k_i ($1 \le i \le n$). Nhiệm vụ của tìm kiếm là xây dựng thuật toán tìm đối tượng có giá trị khóa là X cho trước. X còn được gọi là khóa tìm kiếm hay tham biến tìm kiếm (arrgument).

Công việc tìm kiếm bao giờ cũng hoàn thành bởi một trong hai tình huống:

- Nếu tìm thấy đối tượng có khóa X trong tập các đối tượng thì ta nói phép tìm kiếm thành công (successful).
- Nếu không tìm thấy đối tượng có khóa X trong tập các đối tượng thì ta nói phép tìm kiếm không thành công (unsuccessful).

Trong tìm kiếm, ta luôn hiểu tìm kiếm các đối tượng tổng quát (các bản ghi). Để đơn giản, chúng ta xem mỗi đối tượng như một số trong các thuật toán trình được trình bày tiếp theo.

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.6.1. Tìm kiếm tuyến tính(Sequential Search)

Tìm kiếm đối tượng có giá trị khóa x trong tập các khóa A = {a_1, a_2,.., a_n} là phương pháp so sánh tuần tự x với các khóa {a_1, a_2,.., a_n}. Thuật toán trả lại vị trí của x trong dãy khóa A = {a_1, a_2,.., a_n}, trả lại giá trị -1 nếu x không có mặt trong dãy khóa A = {a_1, a_2,.., a_n}. Độ phức tạp của thuật toán Sequential-Search() là O(n).

Thuật toán Sequential-Search(int A[], int n, int x) {//*tìm x trong dãy A[]*

```
        for (i =0; i<n; i++ ) {
                if ( x ==A[i] )
                        return (i);
        }
        return(-1);
}
```

Ví dụ:

Input :

A[] = { 9, 7, 12, 8, 6, 5}, x = 13.

Output: Sequential-Search(A, n, x) = -1;

Ví dụ:

Input :

A[] = { 9, 7, 12, 8, 6, 5}, x = 6.

Output: Sequential-Search(A, n, x) = 4;

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.6.2. Tìm kiếm nọi suy (Interpolution Search)

Tìm kiếm kiểu nội suy (interpolation search) là thuật toán tìm kiếm giá trị khóa trong mảng đã được đánh chỉ mục theo thứ tự của các khóa. Thời gian trung bình của thuật toán tìm kiếm nội suy là log(log(n)) nếu các phần tử có phân bố đồng đều. Trường hợp xấu nhất của thuật toán là O(n) khi các khóa được sắp xếp theo thứ tự giảm dần.

```
int Interpolation-Search(int A[], int x, int n){  int low = 0,high = n – 1, mid;
        while ( A[low] <= x  &&  A[high] >= x){
                if (A[high] - A[low] == 0)   return (low + high)/2;
                mid = low + ((x - A[low]) * (high - low)) / (A[high] - A[low]);
                if (A[mid] < x) low = mid + 1;
                else if (A[mid] > x)  high = mid - 1;
                else    return mid;
        }
        if (A[low] == x)
                return low;
        return -1;
}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.6.3. Tìm kiếm nhị phân(Binary Search)

Thuật toán tìm kiếm nhị phân là phương pháp định vị phần tử *x* trong một danh sách A[] gồm n phần tử đã được sắp xếp. Quá trình tìm kiếm bắt đầu bằng việc chia danh sách thành hai phần. Sau đó, so sách x với phần từ ở giữa. Khi đó có 3 trường hợp có thể xảy ra:

- **Trường hợp 1**: nếu x bằng phần tử ở giữa A[mid], thì mid chính là vị trí của x trong danh sách A[].
- **Trường hợp 2**: Nếu x lớn hơn phần tử ở giữa thì nếu x có mặt trọng dãy A[] thì ta chỉ cần tìm các phần tử từ mid+1 đến vị trí thứ n.
- **Trường hợp 3**: Nếu x nhỏ hơn A[mid] thì x chỉ có thể ở dãy con bên trái của dãy A[].

Lặp lại quá trình trên cho đến khi cận dưới vượt cận trên của dãy A[] mà vẫn chưa tìm thấy x thì ta kết luận x không có mặt trong dãy A[].

Độ phức tạp thuật toán là : O(log(n)).

Created using easyPDF Printer
Click here to purchase a license to remove this image

```
int Binary-Search( int A[], int n, int x) {//tìm vị trí của x trong dãy A[]
        int  low = 0;//cận dưới của dãy khóa
        int  hight = n-1;//cận trên của dãy khóa
        int  mid = (low+hight)/2; //phần tử ở giữa
        while ( low <=hight) { //lặp trong khi cận dưới vẫn nhỏ hơn cận trên
                if ( x > A[mid] ) //nếu x lớn hơn phần tử ở giữa
                        low = mid + 1; //cận dưới được đặt lên vị trí mid +1
                els if ( x < A[i] )
                        hight = mid -1;//cận trên lùi về vị trí mid-1
                else
                        return(mide); //đây chính là vị trí của x
                mid = (low + hight)/2; //xác định lại phần tử ở giữa
        }
        return(-1); //không thấy x trong dãy khóa A.
}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.6.4. Thuật toán Fibonacci Search

Thuật toán tìm kiếm Fibinacci Search được sử dụng để tìm phần tử x của một mảng đã được sắp xếp với sự hỗ trợ từ các số Fibonacci. Thuật toán xem xét vị trí x với các số Fibonacci để giảm kích cỡ không gian tìm kiếm. Thuật toán sử dụng hiệu quả việc truy nhập vị trí các phần tử của bộ nhớ phụ. Thuật toán có độ phức tạp tính toán là O(log(x)) với x là số cần tìm.

```
int fib[]={0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
        377, 610, 98,1597, 2584, 4181, 6765, 10946, 17711,
        28657, 46368, 75025, 121393, 196418, 317811,
        514229, 832040, 1346269, 2178309, 3524578,
        5702887, 9227465, 14930352, 24157817,
        39088169, 63245986, 102334155, 165580141
 };
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

```
int  fibsearch(int A[], int n, int x){
        //Bước 1. Tìm vị trí k là vị trí cuối cùng để fib[k] >n.
        int inf = 0, pos, k;
        int kk= -1, nn = -1;
        if (nn != n) { k = 0;
                while (fib[k] < n) //xac dinh k la vi tri so Fibonacci nho hon n
                        k++;
        }
        else  k = kk; //k = - 1 và không phải tìm kiếm nữa
        //Bước 2. Tìm vị trí của x trong A dựa vào bước nhảy của số Fib
        while (k > 0) { //lặp nếu k còn lớn hơn 0
                pos = inf + fib[--k]; //pos được tính theo công thức này
                if ((pos >= n) || (x < A[pos]));//không phải àm gì cả
                else if (x > A[pos]){//nếu x lớn hơn A[pos]
                        inf = pos + 1; k--;//tính lại inf và giảm k đi 1
                }
                else
                        return pos; //đây là vị trí cần tìm
        }
        return -1; //x không có mặt trong A[]
}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm thuật toán:

Giả sử dãy A[] = {-100, -50, 2, 3 , 45, 56 ,67 ,78 , 89, 99, 100, 101}.

n=12, x =-50 và x = 67.

Thực hiện thuật toán:

Bước 1. *Tìm k là vị trí cuối cùng để fib[k] >n: k=7 vì fib[7]=13.*

Bước 2 (lặp):

x = 12

Bước	k=?	inf =?	Fib[k]=?	Pos =?
1	6	0	8	8
2	5	0	5	5
3	4	0	3	3
4	3	0	2	2
5	2	0	1	**1**

x = 67

Bước	k=?	inf =?	Fib[k]=?	Pos =?
1	6	0	8	8
2	5	0	5	5
3	3	6	2	8
4	2	6	1	7
5	1	6	1	**7**
6	0	6	0	6

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.6.5. Tìm kiếm đồng dạng (Uniform Search)

Thuật toán Uniform Search là mở rộng của thuật toán Binary Search, trong đó D. Knuth sử dụng một bảng tra cứu để cập nhật các chỉ số các phần tử. Việc làm này tối ưu hơn so với việc xác định điểm giữa của các vòng lặp trong thuật toán tìm kiếm nhị phân.

Thuật toán tạo bảng cho các chỉ số mảng được tiến hành như sau:

```
void make_delta(int n){
        int power = 1;
        int i = 0;
        do {
                int half = power;
                power <<= 1;
                delta[i] = (n + half) / power;
        } while (delta[i++] != 0);
}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

Thuật toán Uniform-Search được tiến hành như sau:

```
int unisearch(int *a, int key){
        int i = delta[0] - 1;  /* vi tri giua mang */
        int d = 0;
        while (1) {
                if (key == a[i]) return i;
                else if (delta[d] == 0) return -1;
                else {
                        if (key < a[i])   i -= delta[++d];
                        else     i += delta[++d];
                }
        }

}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.6.6. Tìm kiếm trên cây nhị phân

Cây nhị phân tìm kiếm được xây dựng dựa vào cấu trúc dữ liệu cây, trong đó nội dung của node thuộc nhánh cây con trái nhỏ hơn nội dung node gốc và nội dung thuộc nhánh cây con bên phải lớn hơn nội dung node gốc. Hai cây con bên trái và bên phải cũng hình thành nên một cây nhị phân tìm kiếm. Chính vì vậy, tìm node trên cây tìm kiếm có độ phức tạp trung bình là O(log (n)). Thuật toán tìm kiếm trên BST được thực hiện như sau:

```
node Search(BST *T, int x) {
        if (T!=NULL ) {
                if (x > T->infor)
                        return (Search(T->right, x));
                else if ( x < T -> infor )
                        return (Search(T->left, x);
                else
                        return(T);
        }
        return(NULL);
}
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.7. Tìm kiếm mẫu (Pattern Searching)

Đối sánh xâu (String matching) là một chủ đề quan trọng trong lĩnh vực xử lý văn bản. Các thuật toán đối sánh xâu được xem là những thành phần cơ sở được cài đặt cho các hệ thống thực tế đang tồn tại trong hầu hết các hệ điều hành. Hơn thế nữa, các thuật toán đối sánh xâu cung cấp các mô hình cho nhiều lĩnh vực khác nhau của khoa học máy tính: xử lý ảnh, xử lý ngôn ngữ tự nhiên, tin sinh học và thiết kế phần mềm.

String-matching được hiểu là việc tìm một hoặc nhiều xâu mẫu (pattern) xuất hiện trong một văn bản (có thể là rất dài). Ký hiệu xâu mẫu hay xâu cần tìm là $X = (x_0, x_1, .., x_{m-1})$ có độ dài m. Văn bản $Y = (y_0, y_1, .., y_{n-1})$ có độ dài n. Cả hai xâu được xây dựng từ một tập hữu hạn các ký tự Alphabet ký hiệu là Σ với kích cỡ là σ. Như vậy một xâu nhị phân có độ dài n ứng dụng trong mật mã học cũng được xem là một mẫu. Một chuỗi các ký tự ABD độ dài m biểu diễn các chuỗi AND cũng là một mẫu.

Input:

- Xâu mẫu X =(x0, x1,.., xm), độ dài m.
- Văn bản Y =(y0, x1,.., yn), độ dài n.

Output:

- Tất cả vị trí xuất hiện của X trong Y.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Phân loại các thuật toán đối sánh mẫu

Thuật toán đối sánh mẫu đầu tiên được đề xuất là Brute-Force. Thuật toán xác định vị trí xuất hiện của X trong Y với thời gian O(m.n). Nhiều cải tiến khác nhau của thuật toán Brute-Force đã được đề xuất nhằm cải thiện tốc độ tìm kiếm mẫu. Ta có thể phân loại các thuật toán tìm kiếm mẫu thành các lớp:

- **Tìm kiếm mẫu từ bên trái qua bên phải**: Harrison Algorithm, Karp-Rabin Algorithm, Morris-Pratt Algorithm, Knuth- Morris-Pratt Algorithm, Forward Dawg Matching algorithm , Apostolico-Crochemore algorithm, Naive algorithm.

- **Tìm kiếm mẫu từ bên phải qua bên trái**: Boyer-Moore Algorithm , Turbo BM Algorithm, Colussi Algorithm, Sunday Algorithm, Reverse Factorand Algorithm, Turbo Reverse Factor, Zhu and Takaoka and Berry-Ravindran Algorithms.

- **Tìm kiếm mẫu từ một vị trí cụ thể**: Two Way Algorithm, Colussi Algorithm , Galil-Giancarlo Algorithm, Sunday's Optimal Mismatch Algorithm, Maximal Shift Algorithm, Skip Search, KMP Skip Search and Alpha Skip Search Algorithms.

- **Tìm kiếm mẫu từ bất kỳ**: Horspool Algorithm, Boyer-Moore Algorithm, Smith Algorithm , Raita Algorithm.

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.7.1. Thuật toán Brute-Force

Đặc điểm:

- Không có pha tiền xử lý.
- Sử dụng không gian nhớ phụ hằng số.
- Quá trình so sánh thực hiện theo bất kỳ thứ tự nào.
- Độ phức tạp thuật toán là O(n.m);

Thuật toán Brute-Force:

Input :

- Xâu mẫu X =(x0, x1,..,xm), độ dài m.
- Văn bàn nguồn Y=(y1, y2,..,yn) độ dài n.

Output:

- Mọi vị trí xuất hiện của X trong Y.

Formats: Brute-Force(X, m, Y, n);

Actions:

```
for ( j = 0; j <= (n-m); j++) { //duyệt từ trái qua phải xâu X
        for (i =0; i<m && X[i] == Y[i+j]; i++) ; //Kiểm tra mẫu
        if (i>=m) OUTPUT (j);
}
```

EndActions.

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.7.2. Thuật toán Knuth-Morris-Pratt

Đặc điểm:

- Thực hiện từ trái sang phải.
- Có pha tiền xử lý với độ phức tạp O(m).
- Độ phức tạp thuật toán là O(n + m);

Thuật toán PreKmp: //thực hiện bước tiền xử lý

Input :

- Xâu mẫu X =(x0, x1,..,xm), độ dài m.

Output: Mảng giá trị kmpNext[].

Formats:

- PreKmp(X, m, kmpNext);

Actions:

```
        i = 1; kmpNext[0] = 0; len = 0; //kmpNex[0] luôn là 0
        while (i < m) {
                if (X[i] == X[len] ) { //Nếu X[i] = X[len]
                        len++;  kmpNext[i] = len; i++;
                }
                else  { // Nếu X[i] != X[len]
                        if ( len != 0 ) {  len = kmpNext[len-1]; }
                        else {  kmpNext[i] = 0; i++; }
                }
        }
```

EndActions.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm PreKmp (X, m, kmpNext):

- X[] = “ABABCABAB”, m = 9.

i=?	(X[i]== X[len])?	Len =?	kmpNext[i]=?
		Len =0	kmpNext[0]=0
i=1	(‘B’==‘A’): No	Len =0	kmpNext[1]=0
i=2	(‘A’==‘A’): Yes	Len =1	kmpNext[2]=1
i=3	(‘B’==‘B’): Yes	Len=2	kmpNext[3]=2
i=4	(‘C’==‘A’): No	Len=0	kmpNext[4]=0
i=5	(‘A’==‘A’): Yes	Len=1	kmpNext[5]=1
i=6	(‘B’==‘B’): Yes	Len=2	kmpNext[6]=2
i=7	(‘A’==‘A’): Yes	Len=3	kmpNext[6]=3
i=8	(‘B’==‘B’): Yes	Len=4	kmpNext[6]=4
Kết luận: kmpNext[] = {0, 0, 1, 2, 0, 1, 2, 3, 4}.			

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm PreKmp (X, m, kmpNext) với X[] = “AABAACAABAA”, m = 11.

i=?	(X[i]== X[len])?	Len =?	kmpNext[i]=?
		Len =0	kmpNext[0]=0
i=1	(‘A’==‘A’): Yes	Len =1	kmpNext[1]=1
i=2	(‘B’==‘A’): No	Len =0	kmpNext[2]=chưa xác định
i=2	(‘B’==‘A’): No	Len=0	kmpNext[2]=0
i=3	(‘A’==‘A’): Yes	Len=1	kmpNext[3]=1
i=4	(‘A’==‘A’): Yes	Len=2	kmpNext[4]=2
i=5	(‘C’==‘B’): No	Len=1	kmpNext[5]=chưa xác định
i=5	(‘C’==‘A’): No	Len=0	kmpNext[5]=chưa xác định
i=5	(‘C’==‘A’): No	Len=0	kmpNext[5]=0
i=6	(‘A’==‘A’): Yes	Len =1	kmpNext[6]=1
i=7	(‘A’==‘A’): Yes	Len =2	kmpNext[7]=2
i=8	(‘B’=‘B’):Yes	Len=3	kmpNext[8] = 3
i=9	(‘A’=‘A’):Yes	Len=4	kmpNext[9] = 4
i=10	(‘A’=‘A’):Yes	Len=5	kmpNext[10] = 5
Kết luận: kmpNext = {0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}			

Created using easyPDF Printer
Click here to purchase a license to remove this image

Thuật toán Knuth-Morris-Partt:

Input :

- Xâu mẫu X =(x0, x1,..,xm), độ dài m.
- Văn bản Y =(y0, y1,..,xn), độ dài n.

Output:

- Tất cả vị trí xuất hiện X trong Y.

Formats: Knuth-Morris-Partt(X, m, Y, n);

Actions:

Bước 1(Tiền xử lý):

preKmp(x, m, kmpNext); //Tiền xử lý với độ phức tạp O(m)

Bước 2 (Lặp):

```
i = 0; j = 0;
while (i < n) {
        if ( X[j] == Y[i] ) { i++; j ++; }
        if ( i == m ) {
                < Tìm thấy mẫu ở vị trí i-j>;
                j = kmpNext[j-1];
        }
        else if (i <n && X[j] != Y[i] ) {
                if (j !=0) j = kmpNext[ j-1];
                else i = i +1;
        }
}
```

EndActions.

Created using easyPDF Printer
Click here to purchase a license to remove this image

Kiểm nghiệm Knuth-Moriss-Patt (X, m, Y, n):

- X[] = “ABABCABAB”, m = 9.
- Y[] = “ABABDABACDABABCABAB”, n = 19

Bước 1 (Tiền xử lý). Thực hiện Prekmp(X, m, kmpNext) ta nhận được:

kmpNext[] = { 0, 0, 1, 2, 0, 1, 2, 3, 4}

Bước 2 (Lặp):

(X[i]==Y[i])?	(J ==9)?	I =? J = ?	
(X[0]==Y[0]): Yes	No	i=1, j=1	
(X[1]==Y[1]): Yes	No	i=2, j=2	
(X[2]==Y[2]): Yes	No	i=3, j=3	
(X[3]==Y[3]): Yes	No	i=4, j=4	
(X[4]==Y[4]): No	No	i=4, j=2	
(X[2]==Y[4]): No	No	i=4, j=0	
(X[0]==Y[4]): No	No	i=5, j=0	
(X[0]==Y[5]): Yes	No	i=6, j=1	
(X[1]==Y[6]): Yes	No	i=7, j=2	
.....			

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.8. Tìm kiếm dựa vào bảng băm (Hash Table)

Ta hình dung các cấu trúc dữ liệu sau được dùng để duy trì thông tin cho một hệ thống thông tin, một cơ sở dữ liệu hay bất kể một ứng dụng nào:

1) Mảng hoặc bản ghi (record).
2) Danh sách liên kết (linked list).
3) Cây tìm kiếm cân bằng.
4) Bảng truy cập trực tiếp (direct acess table).

Với mảng và danh sách liên kết: ta cần thực hiện tìm kiếm với mô hình tuyến tính điều này rất khó thực hiện với các ứng dụng với dữ liệu lớn. Ta có thể sử dụng mảng đã được sắp xếp để tìm kiếm với thời gian O(log(n)), tuy nhiên biện pháp này phải trả giá cho việc duy trì mảng sắp xếp đối với các phép toán insert và delete.

Với cây tìm kiếm cân bằng ta có biện pháp tìm kiếm cỡ O(log(n)). Một giải pháp khác là sử dụng bảng truy cập trực tiếp với thời gian cỡ O(1), tuy nhiên phương pháp này gặp phải một số hạn chế sau:

- Không gian nhớ phụ để lưu trữ bảng truy nhập trực tiếp quá lớn.
- Số nguyên trong ngôn ngữ lập trình là hữu hạn (8byte) nên việc tạo chỉ mục cho các số lớn gặp rất nhiều khó khăn.

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.8.1. Giới thiệu bảng băm (Hash Table)

Do bảng truy nhập trực tiếp không phải lúc nào cũng áp dụng được cho các ứng dụng thực tế, một giải pháp khác được nghiên cứu để thay thế được gọi là bảng băm (hash table) hay còn gọi là ánh xạ băm (hash map).

Hash Table: là một cải tiến của bảng truy cập trực tiếp dựa trên khái niệm hàm băm (hash function). **Hàm băm** là ánh xạ biến đổi các số lớn thành các số nhỏ và sử dụng các số nguyên nhỏ này như chỉ mục trong bảng băm. Bảng băm là thành phần quan trọng thay thế bảng truy cập trực tiếp thực hiện quá trình tìm kiếm. Một Hash Function tốt nếu hàm thỏa mãn hai điều kiện sau:

Có nhiều cách để cài đặt hàm băm:

- Cách thứ nhất là chia dãy khóa thành các đoạn, mỗi đoạn chia dãy khóa theo cùng một nhóm và ghi lại vị trí các đoạn của mỗi nhóm.
- Cách thứ hai là ấn định dãy khóa thành m nhóm, mỗi nhóm được biểu diễn như một danh sách liên kết.
- Cách thứ hai là ấn định dãy khóa thành m nhóm, mỗi nhóm được biểu diễn như một cây nhị phân tìm kiếm.

Created using easyPDF Printer
Click here to purchase a license to remove this image

6.8.2. Ví dụ về cài đặt hàm băm

Định nghía cặp <key, value>:

```
const int TABLE_SIZE = 128;//kích cỡ của bảng băm
class HashEntry {//định nghĩa lớp HashEntry
    public:
        int key; //giá trị khóa
        int value;//giá trị tương ứng với khóa
        HashEntry(int key, int value){//Constructor
            this->key = key;
            this->value = value;
        }
};
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

Định nghía cặp <key, value>:

```
class HashMap{//định nghĩa lớp hàm băm
    private:
        HashEntry **table;//bảng hai chiều các cặp <key, value>
    public:
        HashMap(){//constructor của lớp
            table = new HashEntry * [TABLE_SIZE];
            for (int i = 0; i< TABLE_SIZE; i++){//thiết lập bảng băm là NULL
                table[i] = NULL; //con trỏ này là NULL
            }
        }
        ~HashMap(){//Destructor của hàm băm
            for (int i = 0; i < TABLE_SIZE; i++){
                if (table[i] != NULL)
                    delete table[i]; //giải phóng từng con trỏ
                delete[] table; //giải phóng luôn bảng
            }
        }
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

Định nghía cặp <key, value>:

```
int HashFunc(int key){//đây là hàm băm
        return key % TABLE_SIZE; //tra lai phan du cua key va kich co bang
}
void Insert(int key, int value)      {//thêm một phần tử tương ứng với một khóa
        int hash = HashFunc(key); //lấy giá trị khóa
        while (table[hash] != NULL && table[hash]->key != key){
                hash = HashFunc(hash + 1);
        }
        if (table[hash] != NULL)
                delete table[hash];
        table[hash] = new HashEntry(key, value);
    }
```

Created using easyPDF Printer
Click here to purchase a license to remove this image

```
int Search(int key) {//tìm một phần tử ở vị trí khóa
      int  hash = HashFunc(key); //trước hết lấy vị trí khóa trong bảng băm
      while (table[hash] != NULL && table[hash]->key != key) //tìm vị trí khóa
                hash = HashFunc(hash + 1);
      if (table[hash] == NULL) //nếu khóa không tồn tại
                return -1;
      else
                return table[hash]->value;
}
void Remove(int key){//loai phan tu tai vi tri khoa
       int hash = HashFunc(key);
       while (table[hash] != NULL){
                if (table[hash]->key == key)    break;
                 hash = HashFunc(hash + 1);
           }
           if (table[hash] == NULL) {
              cout<<"Khong co phan tu tai khoa "<<key<<endl;   return;
          }
          else        delete table[hash];
                cout<<"Phan tu da bi loai bo"<<endl;
 }
```

Created using easyPDF Printer
Click here to purchase a license to remove this image