Học viện
Công nghệ Bưu chính Viễn thông

BÀI 1. SẮP XẾP VÀ TÌM KIẾM

CÁC THUẬT TOÁN SẮP XẾP

1. Bài toán sắp xếp
2. Các thuật toán sắp xếp đơn giản
3. Thuật toán Quick-Sort
4. Thuật toán Merge-Sort
5. Thuật toán Radix-Sort

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.
- Để ví dụ, ta xét tập các đối tượng cần sắp xếp là tập các số.

SẮP XẾP ĐƠN GIẢN

Các đặc trưng:

- Ý tưởng dễ hiểu
- Cài đặt đơn giản
- Độ phức tạp cao

Một số thuật toán sắp xếp đơn giản:

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

SẮP XẾP CHỌN (SELECTION SORT)

- Ý tưởng chính: 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.

SẮP XẾP CHỌN (SELECTION SORT) – tiếp

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++) {
    min_idx = i;
    for ( j = i +1; j<n; j++ ) {
        if (Arr[min_idx] > Arr[j] )
            min_idx = j;
    }
    temp = Arr[i] ; Arr[i] = Arr[min_idx]; Arr[min_idx] = temp;
```

End.

SẮP XẾP CHÈN (INSERTION SORT)

- Thuật toán sắp xếp kiểu chèn được thực hiện đơn giản theo cách của người chơi bài thông thường.
  1. Lấy phần tử đầu tiên Arr[0] (quân bài đầu tiên) như vậy ta có dãy một phần tử được sắp.
  2. 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.
  3. 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
  4. Quá trình sắp xếp sẽ kết thúc khi i = n.

SẮP XẾP CHÈN (INSERTION SORT) – tiếp

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++) {
        key = Arr[i];
        j = i-1;
        while (j >= 0 && Arr[j] > key) {
                Arr[j+1] = Arr[j];
                j = j-1;
        }
        Arr[j+1] = key;
}
```

End.

SẮP XẾP NỔI BỌT (BUBBLE SORT)

Thuật toán sắp xếp kiểu nổi bọt thực hiện đổi chỗ hai phần từ liền kề nhau nếu chúng chưa được sắp xếp. Thuật toán dừng khi không còn cặp nào sai thứ tự.

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: Bubble-Sort(Arr, n);

Actions:

```
for (i = 0; i < n; i++) {
        check = false;
        for (j=0; j<n-i-1; j++ ) {
                if (Arr[j] > Arr[j+1] ) {
                        check = true;
                        temp = Arr[j];  Arr[j] = Arr[j+1]; Arr[j+1] = temp;
                }
        }
        if(!check) break;
}
```

End.

SẮP XẾP NHANH (QUICK-SORT)

- Mô hình chia để trị (Devide and Conquer).

- Thuật toán được thực hiện xung quanh 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:
  - Chọn phần tử đầu tiên trong dãy làm chốt.
  - Chọn phần tử cuối cùng trong dãy làm chốt.
  - Chọn phần tử ở giữa dãy làm chốt.
  - Chọn phần tử ngẫu nhiên trong dãy làm chốt.

SẮP XẾP NHANH (QUICK-SORT)

- 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:

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

SẮP XẾP NHANH (QUICK-SORT)

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];    i = (l - 1);
for ( j = l;  j <= h- 1;  j++) {
        if (Arr[j] <= x){
                i++;
                swap(&Arr[i], &Arr[j]);
        }
  }
  swap(&Arr[i + 1], &Arr[h]);
  return (i + 1);
```

End.

SẮP XẾP NHANH (QUICK-SORT)

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) {
        p = Partition(Arr, l, h);
        Quick-Sort(Arr, l, p-1);
        Quick-Sort(Arr, p+1, h);
}
```

End.

SẮP XẾP NHANH (QUICK-SORT)

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};

Cận dưới l=0, cận trên h = 9.

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}	

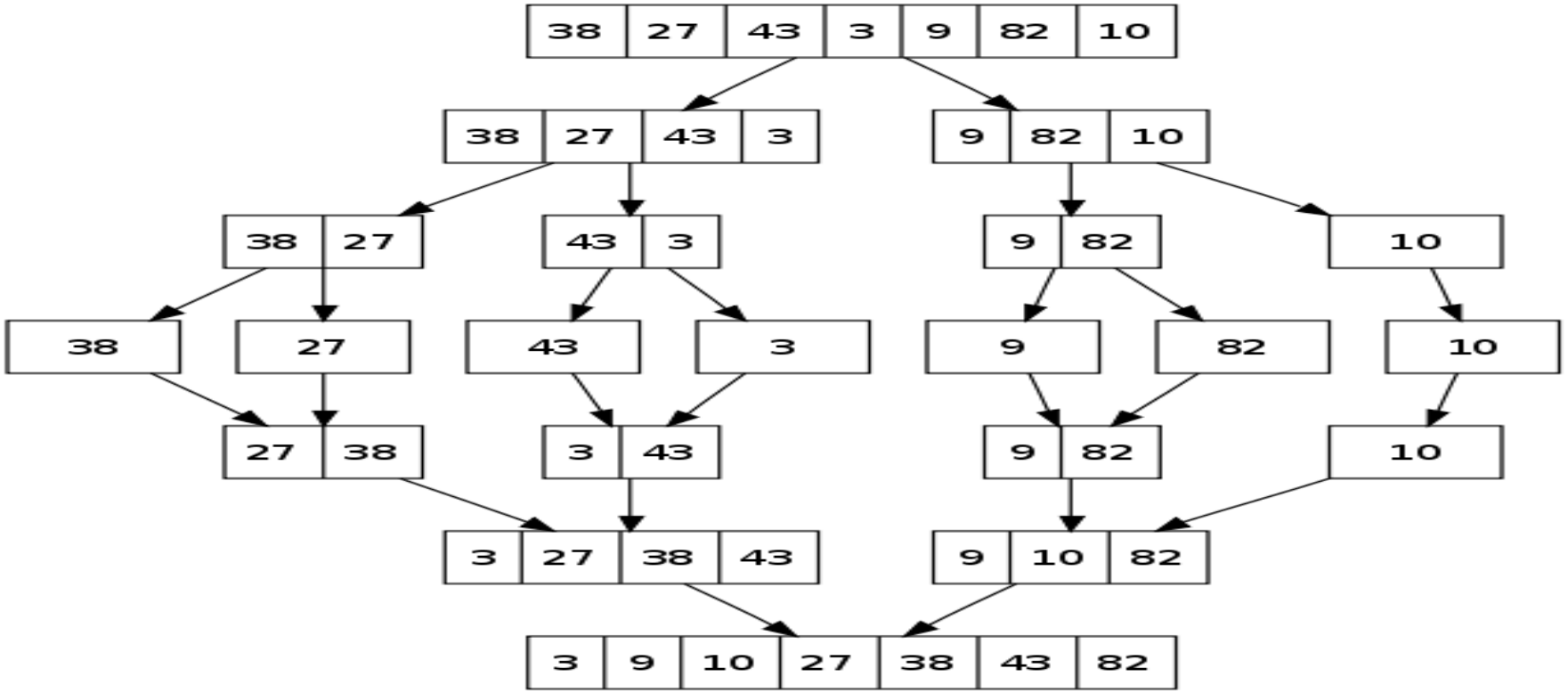
SẮP XẾP TRỘN (MERGE – SORT)

- Giống như Quick-Sort, Merge-Sort cũng được xây dựng theo mô hình chia để trị (Divide 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.
- Xây dựng được một thủ tục hợp nhất (Merge).

SẮP XẾP TRỘN (MERGE – SORT)

Bài toán hợp nhất Merge:

- Cho hai nửa của một dãy Arr[1,..,m] và Arr[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.



SẮP XẾP TRỘN (MERGE – SORT)

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.
L = {20, 22, 24}.
R = {15, 18, 23, 25}.

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)}		

SẮP XẾP TRỘN (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;
        Merge-Sort(Arr, l, m);
        Merge-Sort(Arr, m+1, r);
        Merge(Arr, l, m, r);
}
```

End.

SẮP XẾP TRỘN (MERGE – SORT)

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}

THUẬT TOÁN RADIX-SORT

Giả sử ta cần sắp xếp dãy số nguyên bất kỳ, ví dụ A[] = { 570, 821, 742, 563, 744, 953, 166, 817, 638, 639}.

Ý tưởng: Sắp xếp theo chữ số từ hàng đơn vị trở lên đến hết

Minh họa:

Sắp xếp dãy số theo thứ tự tăng dần của các số hàng đơn vị									
570	821	742	563	953	744	166	817	638	639
Sắp xếp dãy số theo thứ tự tăng dần của các số hàng chục									
817	821	638	639	742	744	953	963	166	570
Sắp xếp dãy số theo thứ tự tăng dần của các số hàng trăm									
166	570	638	639	742	744	817	821	953	963

THUẬT TOÁN RADIX-SORT

PHÂN TÍCH:

- Với hệ cơ số là 10, chỉ có 10 chữ số từ 0 đến 9,
- Sử dụng phương pháp đếm phân bố các số từ 0..9 để thực hiện sắp xếp.
- Sau đó kết hợp với các số hàng chục, hàng trăm...

Thuật toán Radix - Sort phù hợp khi:

- Cơ số của hệ đếm phù hợp với dãy số.
- Việc lấy ra một chữ số là dễ dàng.
- Sử dụng ít lần phép đếm phân phối.
- Phép đếm phân phối thực hiện nhanh

THUẬT TOÁN RADIX-SORT

Cài đặt thuật toán:

```
void radixSort( int[] A)   {
        int i, m = A[0], B[n], exp = 1, n = A.length;
        for (i = 1; i < n; i++) //tìm số lớn nhất trong dãy
                if (A[i] > m)
                        m = A[i];
        while (m / exp > 0) {
                int[] bucket = new int[10];
                for (i = 0; i < n; i++) //đếm phân bố các số từ 0..9
                        bucket[(A[i] / exp) % 10]++;
                for (i = 1; i < 10; i++)
                        bucket[i] += bucket[i - 1];
                for (i = n - 1; i >= 0; i--)
                        B[--bucket[(A[i] / exp) % 10]] = A[i];
                for (i = 0; i < n; i++)
                        A[i] = B[i];
                exp *= 10;
        }
}
```

Học viện
Công nghệ Bưu chính Viễn thông

CÁC THUẬT TOÁN TÌM KIẾM

CÁC THUẬT TOÁN TÌM KIẾM

1. Tìm kiếm tuyến tính
2. Tìm kiếm nhị phân
3. Tìm kiếm tam phân
4. Tìm kiếm nội suy
5. Tìm kiếm kiểu Fibonacci

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.

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

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

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.
- Chia danh sách thành hai phần. So sánh x với phần tử ở giữa.
- Ba trường hợp :
  - **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)).

TÌM KIẾM NHỊ PHÂN (BINARY SEARCH)

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

TÌM KIẾM TAM PHÂN (TERNARY BINARY SEARCH)

- Thuật toán tìm kiếm tam phân là phương pháp định vị phần tử *x* trong một danh sách A[] gồm n phần tử đã được sắp xếp.
- Chia danh sách thành ba phần: {(0, mid1), (mid1+1, mid2), (mid2+1, n)}
  - **Trường hợp 1**: nếu x có giá trị là A[mid1] thì mid1 là kết quả.
  - **Trường hợp 2**: nếu x có giá trị là A[mid2] thì mid2 là kết quả.
  - **Trường hợp 3**: Nếu x nhỏ hơn A[mid1] thì x chỉ có thể ở dãy con bên trái.
  - **Trường hợp 4**: Nếu x lớn hơn A[mid2] thì x chỉ có thể ở dãy con bên phải
  - **Trường hợp 5**: Nếu x nhỏ hơn A[mid2] thì x chỉ có thể ở dãy con đoạn từ [mid1 + 1...mid2].
- 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)).

TÌM KIẾM TAM PHÂN (TERNARY BINARY SEARCH)

```
int ternarySearch (int[] A, int value, int start, int end) {
        if (start > end)   return -1;
        /* Chia dãy thành 3 đoạn: [start..mid1], [mid1..mid2], [mid 2..end]*/
        int mid1 = start + (end-start) / 3;
        int mid2 = start + 2*(end-start) / 3;
        if (A[mid1] == value) return mid1;
        if (A[mid2] == value) return mid2;
        if (value < A[mid1])
                        return ternarySearch (A, value, start, mid1-1);
         if (value > A[mid2])
                        return ternarySearch (A, value, mid2+1, end);
         return ternarySearch (A, value, mid1,mid2);
   }
```

TÌM KIẾM NỘI 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;
}
```

THUẬT TOÁN FIBONACCI SEARCH

Ý tưởng

- Tìm phần tử x của một mảng đã được sắp xếp với sự hỗ trợ từ các số Fibonacci.
- Xem xét vị trí x với các số Fibonacci để giảm kích cỡ không gian tìm kiếm.
- Sử dụng hiệu quả việc truy nhập vị trí các phần tử của bộ nhớ phụ.
- 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
 };
```

THUẬT TOÁN FIBONACCI SEARCH

```
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, kk= -1, nn = -1;
        if (nn != n) { k = 0;
                while (fib[k] < 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) {
                pos = inf + fib[--k];
                if ((pos >= n) || (x < A[pos]));
                else if (x > A[pos]){
                        inf = pos + 1; k--;
                }
                else
                        return pos;
        }
        return -1; //x không có mặt trong A[]
}
```

THUẬT TOÁN FIBONACCI SEARCH

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 = -50

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**