Cấu trúc dữ liệu và giải thuật

Các thuật toán sắp xếp và tìm kiếm

Nguyễn Văn Tiến

Nội dung

1. Giới thiệu bài toán sắp xếp
2. Thuật toán sắp xếp đơn giản
3. Thuật toán sắp xếp nhanh (Quick sort)
4. Thuật toán sắp xếp trộn (Merge sort)
5. Thuật toán sắp xếp theo cơ số (Radix sort)
6. Một số thuật toán tìm kiếm thông dụng

Bài toán sắp xếp

Giới thiệu về bài toán sắp xếp

- Cho dãy gồm n đối tượng $r_0, r_1, .., r_{n-1}$.

- Mỗi đối tượng r_i được tương ứng với một khóa k_i ($0 \leq i < n$).

- 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.

Thuật toán sắp xếp đơn giản

Các đặc trưng

- Ý tưởng đơn giản
- Dễ cài đặt
- Độ phức tạp cao

Một số thuật toán sắp xếp đơn giản

- Thuật toán sắp xếp lựa chọn (Selection sort)
- Thuật toán sắp xếp chèn (Insertion sort)
- Thuật toán sắp xếp 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)

Ví dụ

arr[] = 64 25 12 22 11

11 25 12 22 64

11 12 25 22 64

11 12 22 25 64

11 12 22 25 64

Sắp xếp chọn (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: selectionSort(Arr, n);

Sắp xếp chọn (Selection sort)

```
void selectionSort(int arr[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        swap(arr[min_idx], arr[i]);
    }
}
```

Sắp xếp chèn (Insertion sort)

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

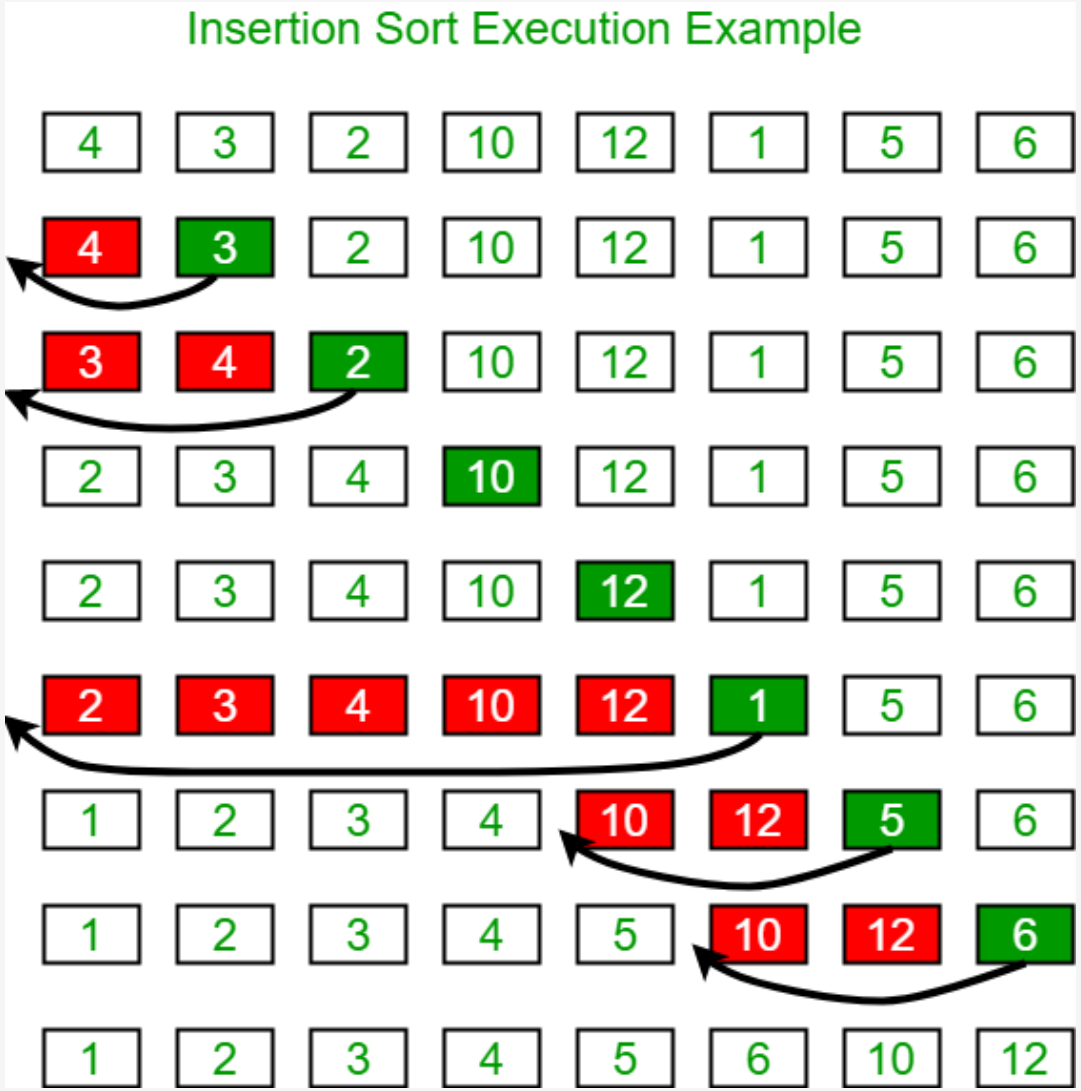
1. Lấy phần tử đầu tiên Arr[0] (quân bài đầu tiên) như vậy ta có dãy một phần tử được sắp.

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 - 1.

Sắp xếp chèn (Insertion sort)



Sắp xếp chèn (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: insertionSort(Arr, n);

Sắp xếp chèn (Insertion sort)

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

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.

Ví dụ

Lượt 1:

(**5 1** 4 2 8) –> (**1 5** 4 2 8)
(1 **5 4** 2 8) –> (1 **4 5** 2 8)
(1 4 **5 2** 8) –> (1 4 **2 5** 8)
(1 4 2 **5 8**) –> (1 4 2 **5 8**)

Sắp xếp nổi bọt (Bubble sort)

Lượt 2:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

Lượt 3:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

Sắp xếp nổi bọt (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: bubbleSort(Arr, n);

Sắp xếp nổi bọt (Bubble sort)

```
void bubbleSort(int arr[], int n) {
    int i, j;
    bool swapped;
    for (i = 0; i < n-1; i++)
    {
        swapped = false;
        for (j = 0; j < n-i-1; j++)
        {
            if (arr[j] > arr[j+1])
            {
                swap(arr[j], arr[j+1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

Sắp xếp nhanh (Quick sort)

Mô hình chia để trị (Divide 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 low và kết thúc tại high.
- Cận dưới của dãy con: low
- Cận trên của dãy con: high

Output:

- Vị trí chính xác của arr[high] nếu dãy arr[] được sắp xếp.

Formats: partition(arr, low, high);

Sắp xếp nhanh (Quick sort)

```
int partition (int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}
```

10, 27, 15, 29, 21, 11, 14, 18, 12, 17

10, 15, 11, 14, 12, **17**, 29, 18, 21, 27

Sắp xếp nhanh (Quick sort)

Input :

Dãy Arr[] gồm n phần tử.

Cận dưới của dãy: low.

Cận trên của dãy : high

Output:

Dãy Arr[] được sắp xếp.

Formats: quickSort(Arr, low, high);

Sắp xếp nhanh (Quick sort)

```
void quickSort(int arr[], int low, int high) {
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Sắp xếp nhanh (Quick sort)

Độ phức tạp thuật toán:

Trường hợp xấu nhất: $O(n^2)$.

Trường hợp tốt nhất : O(nlog(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 low = 0, cận trên high = 9.

Sắp xếp nhanh (Quick sort)

p = partition(Arr,low,high)	Giá trị Arr[]=?
p=5: low=0, high=9	{10,15,11,14,12}, **(17)**,{29,18, 21, 27}
P=2: low=0, high=4	{10,11},**(12)**, {14,15}, (17),{29,18, 21, 27}
P=1: low=0, high=1	{10,**11**},(12), {14,15}, (17),{29,18, 21, 27}
P=4: low=3, high=4	{10,11},(12), {14,**15**}, (17),{29,18, 21, 27}
P=8: low=6, high=9	{10,11},(12), {14,15}, (17),{18,21},(**27**),{29}
P=7: low=6, high=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 chia 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).

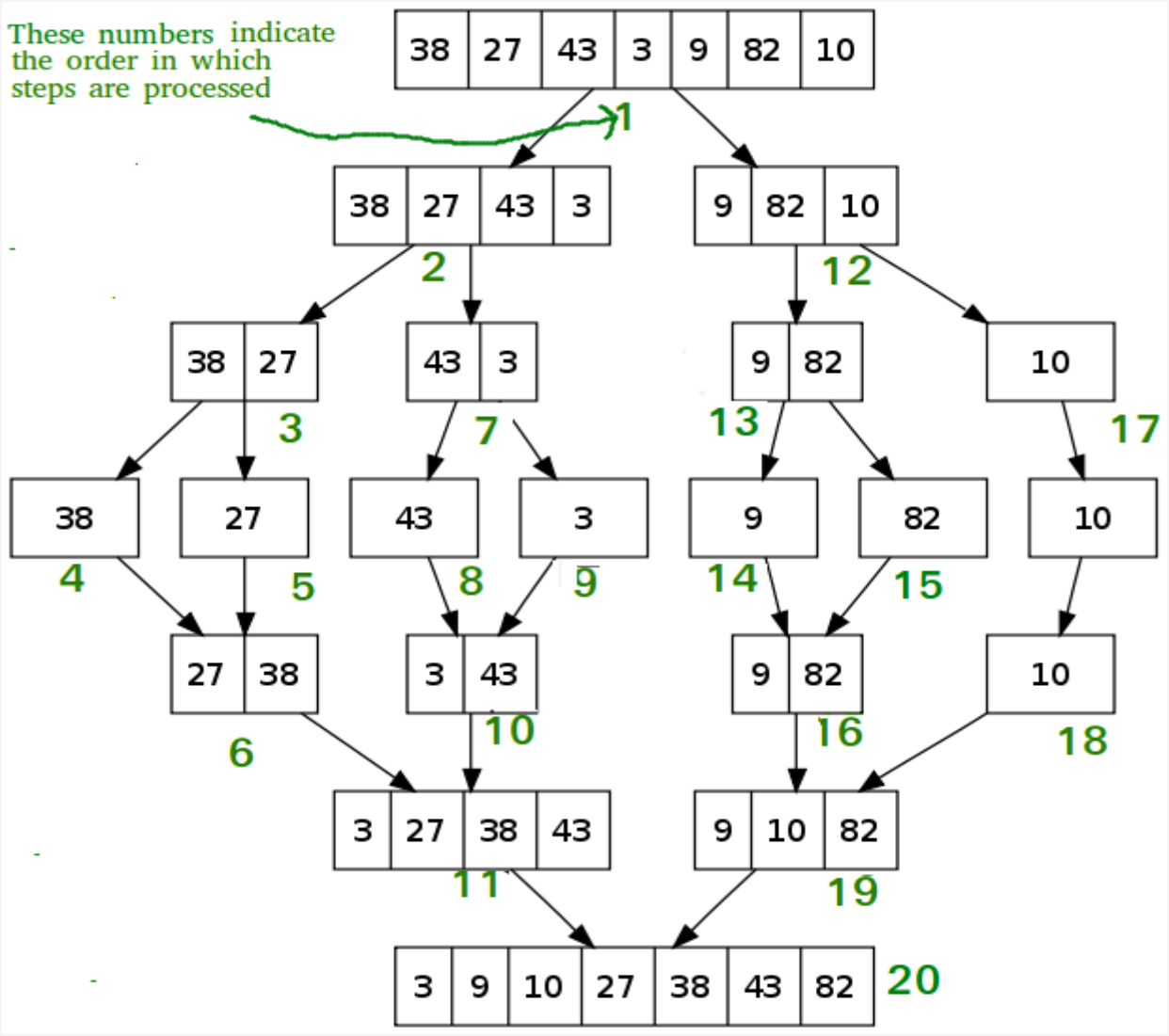
Sắp xếp trộn (Merge sort)

Bài toán hợp nhất Merge:

Cho hai nửa của một dãy Arr[l,..,m] và Arr[m+1,..,r] đã được sắp xếp.

Nhiệm vụ của ta là hợp nhất hai nửa của dãy Arr[l,..,m] và Arr[m+1,..,r] để trở thành một dãy Arr[l, 2,..,r] cũng được sắp xếp.

Sắp xếp trộn (Merge sort)



Sắp xếp trộn (Merge sort)

```
void merge(int arr[], int l, int m, int r) {
    int i, j, k; int n1 = m - l + 1; int n2 = r - m; int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    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++;
    }
}
```

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= 4.

L = {20, 22, 24}.

R = {15, 18, 23, 25}.

Sắp xếp trộn (Merge sort)

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: r;

Output:

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

Formats: mergeSort(Arr, l, r);

Sắp xếp trộn (Merge sort)

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Sắp xếp trộn (Merge sort)

Độ phức tạp thuật toán: O(nlog n).

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

Input :

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

Sắp xếp trộn (Merge sort)

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}

Sắp xếp theo cơ số (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									
57**0**	82**1**	74**2**	56**3**	95**3**	74**4**	16**6**	81**7**	63**8**	63**9**
Sắp xếp dãy số theo thứ tự tăng dần của các số hàng chục									
8**17**	8**21**	6**38**	6**39**	7**42**	7**44**	9**53**	9**63**	1**66**	5**70**
Sắp xếp dãy số theo thứ tự tăng dần của các số hàng trăm									
166	**570**	**638**	**639**	**742**	**744**	**817**	**821**	**953**	**963**

Sắp xếp theo cơ số (Radix sort)

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

Sắp xếp theo cơ số (Radix sort)

```
void countSort(int arr[], int n, int exp)
{
    int output[n];
    int i, count[10] = { 0 };
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixsort(int arr[], int n)
{
    int m = getMax(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

MỘT SỐ THUẬT TOÁN TÌM KIẾM THÔNG DỤNG

Bài 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

Bài toán tìm kiếm

Cho dãy gồm n đối tượng r_0, r_1, .., r_{n-1}. Mỗi đối tượng ri được tương ứng với một khóa ki ($0 \leq i < n$).

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_0, a_1,.., a_{n-1}} là phương pháp so sánh tuần tự x với các khóa {a_0, a_1,.., a_{n-1}}.

Thuật toán trả lại vị trí của x trong dãy khóa A = {a_0, a_1,.., a_{n-1}}, trả lại giá trị -1 nếu x không có mặt trong dãy khóa A = {a_0, a_1,.., a_{n-1}}.

Độ phức tạp của thuật toán Sequential-Search() là O(n).

```
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 từ 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[].

Tìm kiếm nhị phân (Binary search)

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 BinarySearch(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[mid])
            high = mid -1;
        else
            return mid;
        mid = (low + high) / 2;
    }
    return -1;
}
```

Tìm kiếm tam phân (Ternary 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[].

Tìm kiếm tam phân (Ternary search)

```
int ternarySearch (int[] A, int value, int start, int end) {
    if (start > end)    return -1;
     /* Chia day: [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.

Tìm kiếm nội suy (Interpolution search)

```
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;
}
```