Thuật toán tìm kiếm

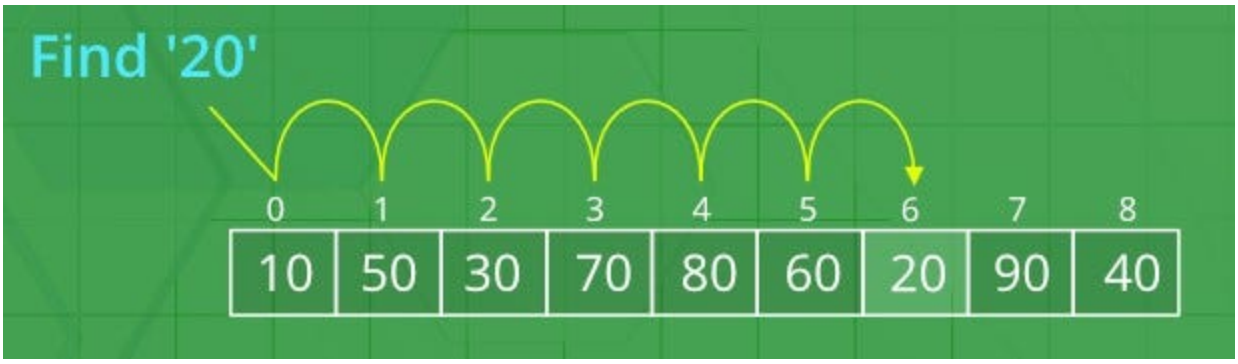
5 Thuật Toán Tìm Kiếm Mọi LTV Nên Biết

Bài viết này sẽ trình bày ngắn gọn về năm thuật toán tìm kiếm cùng với việc triển khai chúng trong C ++ và Java.

- Thuật toán Linear Search
- Thuật toán Binary Search
- Thuật toán Ternary Search
- Thuật toán Jump Search
- Thuật toán Exponential Search

Thuật toán Linear Search

Đây là thuật toán đơn giản nhất trong tất cả các thuật toán tìm kiếm. Trong kiểu tìm kiếm này, một hoạt động tìm kiếm liên tiếp được diễn ra qua tất cả từng phần tử. Mỗi phần tử đều được kiểm tra và nếu tìm thấy bất kỳ kết nối nào thì phần tử cụ thể đó được trả về; nếu không tìm thấy thì quá trình tìm kiếm tiếp tục diễn ra cho tới khi tìm kiếm hết dữ liệu.



Độ phức tạp về thời gian: O (n)

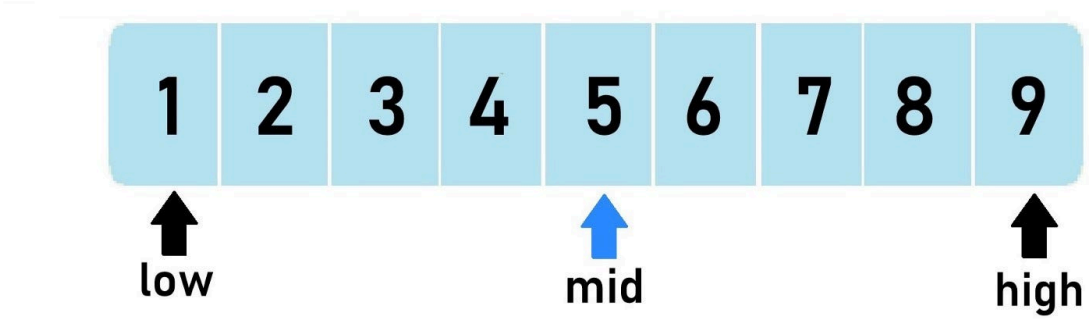
Độ phức tạp không gian: O (1)

Dưới đây là một ví dụ việc triển khai tìm kiếm tuyến tính trong C ++:

```
#include <iostream>
#include <array>
using namespace std;

int linearSearch(int arr[], int target, int n)
{
   for( int i = 0; i < n; i++ ) {
      if ( arr[i] == target) {
         return i;
      }
   }
   return -1;
}

int main()
{
   int arr[] = {13, 42, 41, 61, 26};
   int target = 422;
   int n = sizeof(arr)/sizeof(a[0]);
   int res = linearSearch(arr, target, n);
   cout << "The result of linear search = " << res << endl;
   return 0;
}
```

Thuật toán Binary Search

Binary Search - tìm kiếm nhị phân, còn gọi là tìm kiếm nửa khoảng, tìm kiếm logarit, hay binary chop, là một thuật toán tìm kiếm xác định vị trí của một giá trị cần tìm trong một mảng đã được sắp xếp. Thuật toán tiến hành so sánh giá trị cần tìm với phần tử đứng giữa mảng.



Độ phức tạp thời gian:

O (log [n]) trong đó cơ số của log = 2

Độ phức tạp của không gian: O (1) để thực hiện lặp trong khi O (log [n]) để thực hiện đệ quy vì với mỗi lần gọi đệ quy, một ngăn xếp mới được tạo ra.

Dưới đây là cách triển khai đệ quy của tìm kiếm nhị phân trong Java. Để thực hành, hãy cố gắng tự viết bản triển khai lặp đi lặp lại.

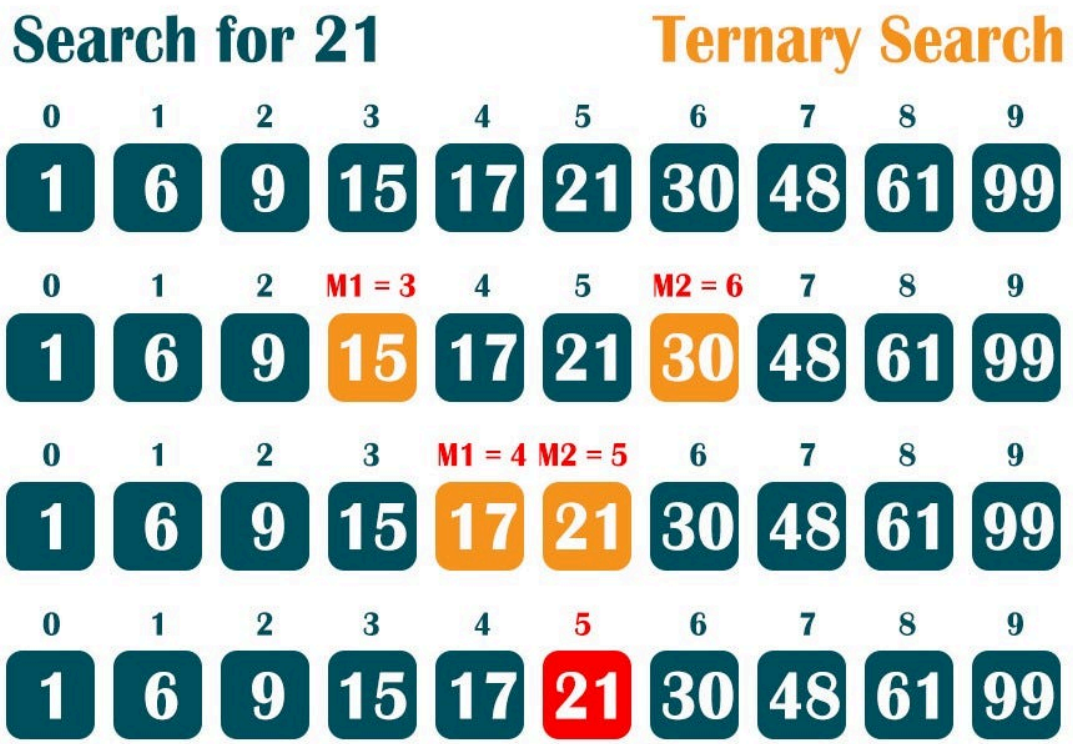
Ví dụ:

```
public int binarySearchRecursive( int a[], int l, int r, int target)
   {
      // RECURSIVE IMPLEMENTATION OF BINARY SEARCH
      int mid = (l+r)/2;

      // base case 1 = checks if array is empty or not
      if ( r - l < 0) {
         return -1;
      }
      // base case 2 = checks if array middle is the target value
      if ( target == a[mid] ) {
         return mid;
      }
      // recursive case 1 = removes left array
      else if ( target >= a[mid]) {
```

```
            return binarySearchRecursive(a, mid+1, r, target);
        }
        // recursive case 2 = removes right array
        else {
            return binarySearchRecursive(a, l, mid-1, target);
        }
    }
```

Thuật toán Ternary Search

Tương tự với thuật toán tìm kiếm nhị phân, Ternary Search - Tìm kiếm tam phân là một kỹ thuật trong khoa học máy tính dùng để tìm kiếm giá trị lớn nhất (maximum) hay nhỏ nhất (minimum) của một unimodal function, và đây cũng là một ví dụ ứng dụng lớp thuật toán Chia để trị (divide and conquer).



Độ phức tạp thời gian: O (log [n]) trong đó cơ số của log = 3

Độ phức tạp của không gian: O (1) để thực hiện lặp lại trong khi O (log [n]) để thực hiện đệ quy

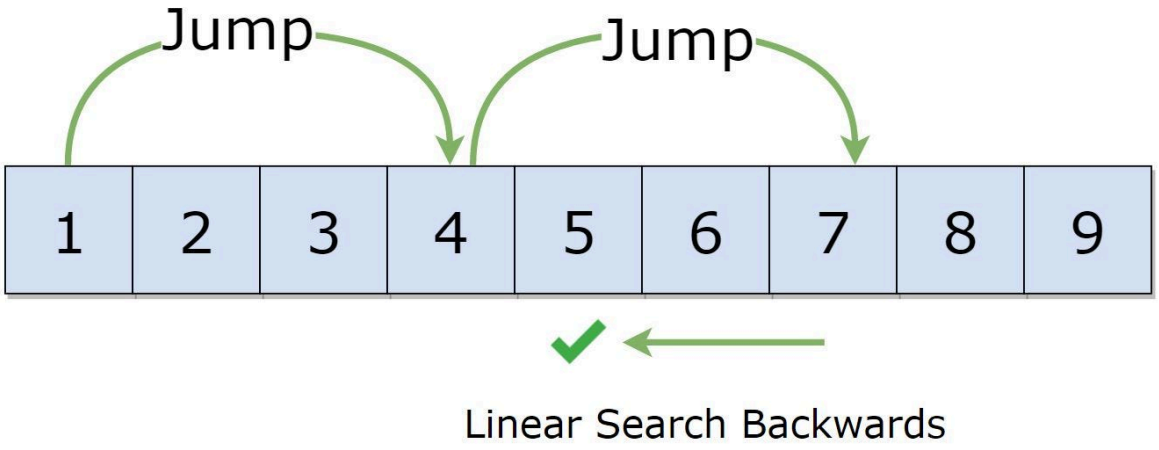
Ví dụ:

```
#include<iostream>
using namespace std;

int ternaryI(int a[], int target, int n)
{
    int l = 0;
    int r = n-1;

    while( r-l>=0 ) {
        int partiton = (r-l)/3;
        int mid1 = l + partiton;
        int mid2 = r - partiton;

        if ( target == a[mid1])
            return mid1;
        else if ( target == a[mid2])
            return mid2;
        else if ( target < a[mid1] )
            r = mid1-1;
        else if ( target > a[mid2] )
            l = mid2+1;
        else {
            l = mid1+1;
            r = mid2-1;
        }
    } // while ends

    return -1;
} // function ends
```

```
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = 10;
    int target = 7;

    int tsi = ternaryI(arr, target, n);
    cout << "Result of TSI = " << tsi << endl;

    return 0;
} // main ends
```

Thuật toán Jump Search

Cơ chế của **Jump Search** đó là tìm ra một hệ số nhảy được tính bằng : Căn bậc hai của số phần tử. Từ hệ số tìm được, **Jump Search** sẽ thực hiện nhảy phần tử theo hệ số để tìm ra phần từ lớn hơn giá trị tìm kiếm. ⇒ Phần tử tìm kiếm sẽ nằm trong khoảng của nhảy mà chứa phần từ lớn hơn giá trị tìm kiếm ở trên.



Độ phức tạp về thời gian: O (log [sqrt (n)])

Độ phức tạp của không gian: O (1) để triển khai lặp lại trong khi O (log [sqrt (n)]) để triển khai đệ quy

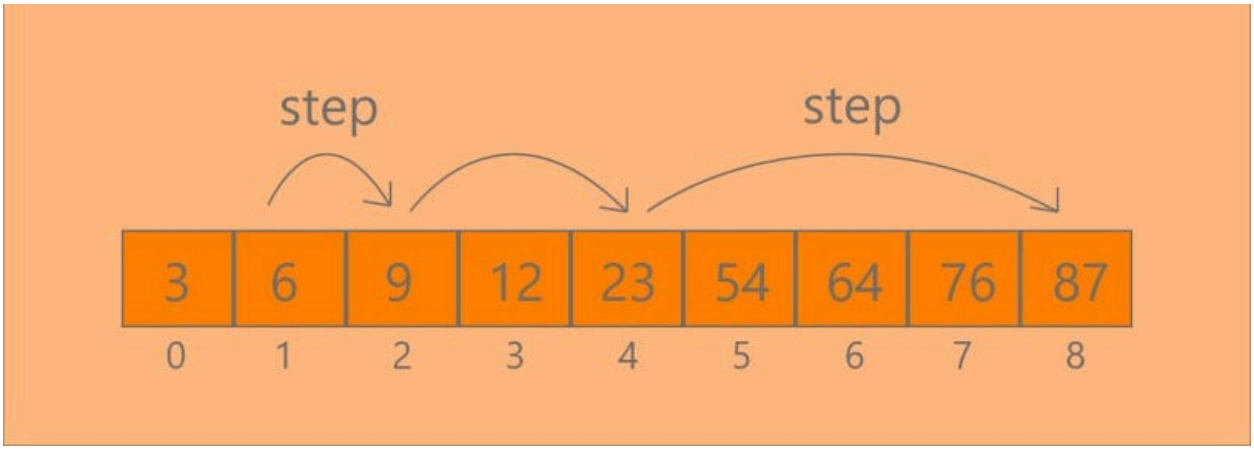
Ví dụ:

```
// Iterative
  public int jumpSearchIterative(int[] a, int target)
  {
     int blockSize = (int) Math.sqrt(a.length);
     int start = 0;
     int next = blockSize;

     while( start < a.length && target > a[next-1] )
     {
        start = next;
        next = next + blockSize;

        if ( next >= a.length )
           next = a.length;
     }

     for( int i=start; i<next; i++ )
     {
        if ( target == a[i])
           return i;
     }

     return -1;


  } // method ends
```

Thuật toán Exponential Search

Exponential Search là một cải tiến so với tìm kiếm nhị phân. Nó hoạt động trên một mảng được sắp xếp nhất định. Thay vì thực hiện tìm kiếm nhị phân trên toàn bộ tập dữ liệu, chúng ta tìm một khối có giá trị đích và sau đó thực hiện tìm kiếm nhị phân trong khối nhỏ đó. Để tìm khối, chúng ta lấy chỉ số (1) và kiểm tra giá trị của

nó với giá trị đích. Nếu mục tiêu nhiều hơn giá trị chỉ mục này, thì chúng ta nhân đôi độ dài của khối hiện tại và kiểm tra chỉ mục (2) và tiếp tục làm điều này cho chỉ mục (4), (8), (16), v.v. cho đến khi tìm ra khối thích hợp. Vì toàn bộ hoạt động của khối này xảy ra trong 1–2–4–8–16-..., đó là lý do tìm kiếm này được gọi là tìm kiếm theo cấp số nhân.



Độ phức tạp về thời gian: O (log [i]) trong đó i là chỉ số của phần tử cần tìm kiếm

Độ phức tạp không gian: O (1) khi chúng ta sử dụng lặp đi lặp lại tìm kiếm nhị phân trong khi O (log [i]) để thực hiện đệ quy tìm kiếm nhị phân.

Ví dụ:

```
package search;

public class Exponential extends Binary
{
   public int expoSearch( int[] a, int target)
   {
      int bound = 1;
      while ( bound < a.length && target > a[bound])
         bound *= 2;

//    This below while loop does the same job
//    while ( bound < a.length) {
//       if ( target <= a[bound] ) {
//          break;
```

```
//        }
//        else {
//            bound = bound * 2;
//        }
//    }

        int left_bound = bound/2;
        int right_bound = Math.min(bound, a.length-1);
        int answer = binarySearchRecursive(a, left_bound, right_bound, target);
        return answer;
    }
}
```

Kết luận

5 thuật toán này không phải là những thuật toán tìm kiếm duy nhất hiện có. Còn một số thuật toán tìm kiếm phổ biến khác như Tìm kiếm Nội suy hay Tìm kiếm Fibbonaci. Nhưng nếu bạn chú ý quan sát, bạn sẽ nhận ra tìm kiếm bậc ba, bước nhảy và hàm mũ chẳng qua là một bản sửa đổi của tìm kiếm tuyến tính hoặc nhị phân. Và trường hợp của phép nội suy và tìm kiếm fibonacci cũng vậy.

Do đó, nếu bạn thành thạo với tìm kiếm tuyến tính và nhị phân, bạn sẽ dễ dàng nắm bắt được bất kỳ thuật toán tìm kiếm nào trong số này. Chúc các bạn học tốt!