Thuật toán tìm kiếm

Module: Advanced Programming with JavaScript

Mục tiêu

- Trình bày được thuật toán tìm kiếm tuyến tính
- Cài đặt được thuật toán tìm kiếm tuyến tính
- Trình bày được thuật toán tìm kiếm nhị phân
- Cài đặt được thuật toán tìm kiếm nhị phân
- Trình bày được độ phức tạp của thuật toán
- Tính được độ phức tạp của thuật toán cho những trường hợp thông dụng

Thuật toán tìm kiếm

- Thao tác tìm kiếm (searching) là một tác vụ thường được sử dụng trong các hệ lưu trữ và quản lý dữ liệu
- Do dữ liệu lớn nên tìm ra giải thuật tìm kiếm nhanh chóng là mối quan tâm hàng đầu.
  - Để đạt được điều này dữ liệu phải được tổ chức theo một thứ tự nào đó thì việc tìm kiếm sẽ nhanh chóng và hiệu quả hơn.
- Các giải thuật tìm kiếm trên danh sách là loại giải thuật cơ bản nhất.
  - Mục đích là tìm trong một tập hợp một phần tử chứa một khoá nào đó.
- Có 2 giải thuật tìm kiếm thường được áp dụng
  - Tìm kiếm tuyến tính (linear search) thường được thực hiện với mảng hay danh sách chưa được sắp xếp thứ tự.
  - Tìm kiếm nhị phân (binary search) thường được thực hiện với mảng hay danh sách đã sắp xếp thứ tự.

Tìm kiếm tuyến tính

Linear search

Tìm kiếm tuyến tính

- Tìm kiếm tuyến tính là hoạt động tìm kiếm liên tiếp được diễn ra qua tất cả các 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 tìm thấy thì quá trình tìm kiếm tiếp tục diễn ra cho tới khi tìm kiếm hết dữ liệu.
- Tìm kiếm tuyến tính còn được gọi là tìm kiếm tuần tự (Sequential searching)

Tìm kiếm tuyến tính

- Ví dụ: Tìm phần tử có giá trị 33

- **Lưu ý:** Giải thuật này chỉ hiệu quả khi cần tìm kiếm trên một mảng/danh sách đủ nhỏ hoặc một mảng/danh sách chưa sắp thứ tự.

Tìm kiếm tuyến tính

- Thuật toán tìm kiếm tuyến tính trên một mảng/danh sách

```
Bắt đầu phương thức linearSearch (list, value)

   for mỗi phần tử trong danh sách
        if match item == value
             return vị trí của phần tử
        kết thúc if
   kết thúc for
   return -1

Kết thúc phương thức
```

Tìm kiếm tuyến tính

- Cài đặt

```
function linearSearch(arr: number[], value: number)
{
    for (let i = 0; i < arr.length; i++)
    {
        if (arr[i] == value)
            return i;
    }
    return -1;
}
```

Tìm kiếm nhị phân

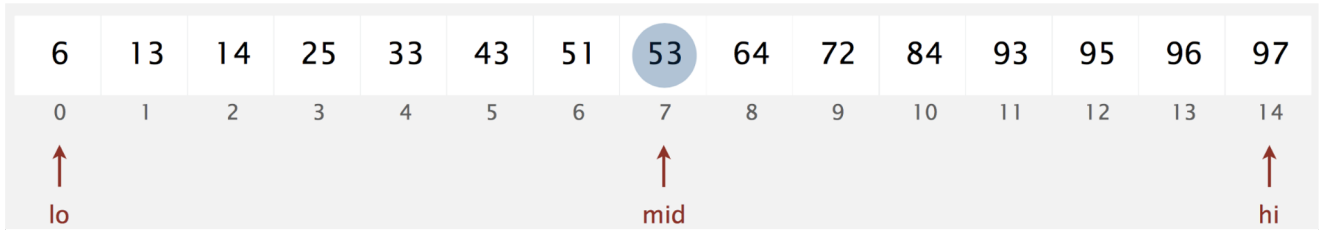
Binary search

Tìm kiếm nhị phân

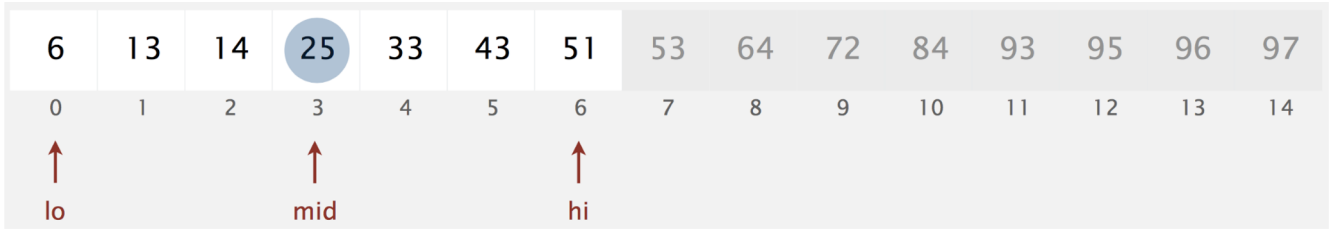
- Tìm kiếm nhị phân được thực hiện trên mảng đã được sắp xếp
- Tìm kiếm một phần tử cụ thể bằng cách so sánh phần tử tại vị trí giữa nhất của tập dữ liệu.
  - Nếu tìm thấy kết nối thì chỉ mục của phần tử được trả về.
  - Nếu phần tử cần tìm là lớn hơn giá trị phần tử giữa thì phần tử cần tìm được tìm trong mảng con nằm ở bên phải phần tử giữa;
  - Nếu không thì sẽ tìm ở trong mảng con nằm ở bên trái phần tử giữa.
  - Tiến trình sẽ tiếp tục như vậy trên mảng con cho tới khi tìm hết mọi phần tử trên mảng con này.

- Ví dụ: Tìm phần tử có giá trị 33

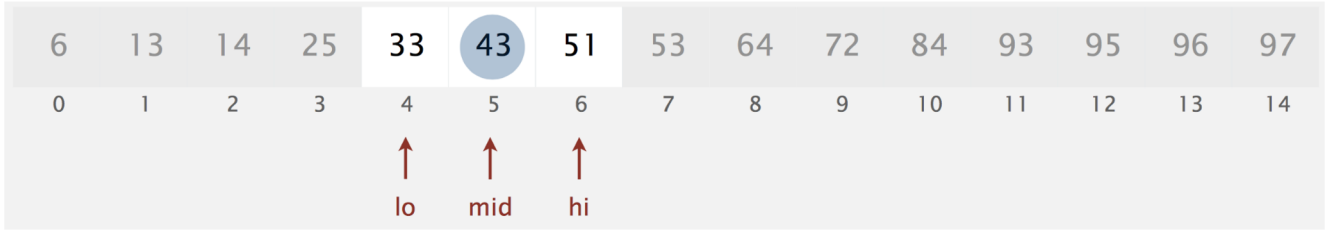
1



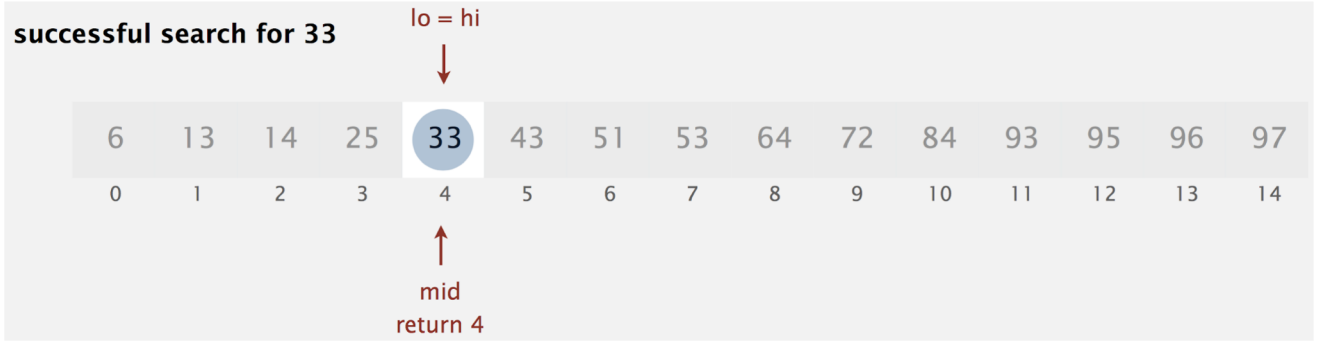
2



3



4



Tìm kiếm nhị phân

- Tìm kiếm nhị phân trên một danh sách đã sắp xếp

```
Bắt đầu phương thức binarySearch (list, value)

   low = 0; high = list.lengh-1;
   while (high >= low)
   bắt đầu while
       mid = (low + high)/2;
       if a[mid] = value: Tìm thấy. Trả về vị trí mid
       else if a[mid] > value, tìm tiếp trong value trong dãy con a[low] ... a[mid-1]
           high = mid – 1
       else if a[mid] < value, tìm tiếp trong value trong dãy con a[mid + 1] ... a[high]
         low = mid + 1
   kết thúc while
   return -1

Kết thúc phương thức
```

Tìm kiếm nhị phân

- Thuật toán tìm kiếm nhị phân sử dụng đệ quy

```
Bắt đầu phương thức binarySearch (a[], low, high value)

    low = 0; high = list.lengh-1; mid = (low + high)/2;
    if ( high >= low)
        if a[mid] = value  return mid;
        if a[mid] > value
        return binarySearch(a, low, mid-1, value);
        if a[mid] < value
        return binarySearch(a, mid + 1, high, value);
    kết thúc if
    return -1

Kết thúc phương thức
```

Tìm kiếm nhị phân

- Cài đặt

```
function binarySearch(arr: number[],low: number,high:
number, value: number): number {
    if (high>=low) {
        const mid = low + (high - low)/2;
        if (arr[mid] == value)
          return mid;
        if (arr[mid] > value)
          return binarySearch(arr, low, mid-1, value);
        return binarySearch(arr, mid+1, high, value);
    }
    return -1;
}
```

Độ phức tạp thuật toán

Độ phức tạp thuật toán

Đánh giá độ phức tạp thuật toán

Tính độ phức tạp những trường hợp thông dụng

Độ phức tạp thuật toán (1)

- Thời gian máy tính thực hiện một thuật toán phụ thuộc vào:
  - Bản thân thuật toán
  - Cấu hình máy tính thực thi thuật toán
- Để đánh giá hiệu quả của một thuật toán có thể xét dựa trên
  - **Thời gian thực hiện thuật toán:** Được đánh giá bằng việc tính số phép tính chính như phép so sánh, phép lặp ...
  - **Không gian cần thiết để thực hiện thuật toán:** Được hiểu là các yêu cầu về bộ nhớ, thiết bị lưu trữ .. của máy tính để thuật toán có thể làm việc.
- Thông thường, các phép tính được thực hiện phụ thuộc vào độ lớn đầu vào.
  - Thời gian thực hiện thuật toán còn được hiểu là một hàm phụ thuộc đầu vào T = f(input)

Độ phức tạp thuật toán (2)

Độ phức tạp thuật toán được hiểu là thời gian thực hiện số phép tính mà thuật toán cần thực hiện với bộ dữ liệu đầu vào có kích thước **n**: T = f(n)

- n có thể là số phần tử của mảng trong trường hợp bài toán sắp xếp hoặc tìm kiếm,
- n là độ lớn của số như trong bài toán kiểm tra số nguyên tố

Thời gian thực hiện giải thuật

- Thời gian chạy trong trường hợp xấu nhất (worse-case running time) là thời gian chạy lớn nhất của thuật toán đó trên tất cả các dữ liệu cùng cỡ.
- Thời gian chạy trung bình là trung bình cộng thời gian chạy trên tất cả các bộ dữ liệu cùng cỡ
- Thời gian chạy trong trường hợp tốt nhất (best-case running time) là thời gian chạy ít nhất của thuật toán đó trên tất cả các dữ liệu cùng cỡ.

Ý nghĩa độ phức tạp thuật toán

- So sánh giữa các thuật toán để biết được:
  - Thuật toán nào tốt hơn
  - Thuật toán nào nhanh hơn
  - Thuật toán nào ít tốn bộ nhớ hơn

Đánh giá độ phức tạp thuật toán

- Đánh giá độ phức tạp của một thuật toán dùng 2 ký hiệu O-lớn (Big-O) và Theta (Θ), thường sử dụng O-lớn hơn.
- Thuật toán A có thời gian thực hiện là T(n) = O(f(n)). Khi đó, thuật toán A có độ phức tạp f(n).
- Ví dụ:
  - Thời gian thực hiện T(n) của chương trình là O(n2), có nghĩa là tồn tại các hằng số dương C và n0 sao cho T(n) <= Cn2 với n >= n0.

Quy tắc tính độ phức tạp

- Quy tắc bỏ hằng số
  - T(n) = O(Cf(n)) = O(f(n)) với C là một hằng số dương.
- Quy tắc cộng
  - 2 chương trình P1 và P2 nối tiếp nhau. T1(n) = O(f(n)) là thời gian thực hiện chương trình 1 và T2(n) = O(g(n)) là thời gian thực hiện chương trình 2.
  - Thời gian thực hiện tổng thể 2 chương trình này là:

    $$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$
- Quy tắc nhân
  - 2 chương trình P1 và P2 lồng nhau. T1(n) = O(f(n)) là thời gian thực hiện chương trình P1 và T2(n) = O(g(n)) là thời gian thực hiện chương trình P2.
  - Thời gian thực hiện tổng thể 2 chương trình này là:

    $$O(f(n).g(n)).$$

Tính độ phức tạp câu lệnh đơn giản

- Các câu lệnh đơn giản
  - Các câu lệnh đơn giản như gán, so sánh, return ... có thời gian thực hiện là O(1)
  - Ví dụ: m = 0 thời gian thực hiện là O(1) hay còn gọi là độ phức tạp hằng số

Tính độ phức tạp vòng lặp

- Vòng lặp
  - Trong phần lớn các trường hợp, thời gian thực hiện vòng lặp bằng thời gian thực hiện thân vòng lặp n lần
- Ví dụ 1:

```
for (let i = 1; i <= n; i++) {
  k = k + 5;
}
```

Vòng lặp trên thực hiện n lần
Số phép tính thực hiện là O(1) là hằng số C: k = k + 5
Vậy tổng thời gian thực hiện T(n) = O(Cn) = O(n).

Tính độ phức tạp các trường hợp thông dụng

- Ví dụ 2:

```
for (let i = 1; i <= n; i++) {
  for (let j = 1; j <= n; j++) {
    k = k + i + j;
  }
}
```

Vòng lặp ngoài và trong cùng thực hiện n lần
Số phép tính thực hiện là O(1) là hằng số C: k = k + i + j

Vậy tổng thời gian thực hiện T(n) = O(Cn*n) = O(n2).

Tính độ phức tạp các trường hợp thông dụng

- Ví dụ 3:

```
for (let i = 1; i <= n; i++) {
  for (let j = 1; j <= 20; j++) {
    k = k + i + j;
  }
}
```

Vòng lặp trong thực hiện 20 lần, vòng lặp ngoài thực hiện n lần.
Số phép tính thực hiện là O(1) là hằng số C: k = k + i + j

Vậy tổng thời gian thực hiện T(n) = O(C*20*n) = O(n).

Tính độ phức tạp cấu trúc rẽ nhánh

- Lệnh rẽ nhánh
  - Tổng thời gian thực hiện lớn nhất = Thời gian kiểm tra điều kiện if (thường là hằng số) + thời gian thực hiện then + thời gian thực hiện else.
- Ví dụ:

```
if (a < 0) {
    m = 0;
} else {
    for (let i = 0; i < n; i++) {
            m += 1;
    }
}
```

  - Số phép tính thực hiện: a < 0: C1; m = 0: C2; m+=1: C3
  - Tổng thời gian thực hiện T(n) = (C1 + C2 + C3)*n = O(n)

Tổng kết

- Tìm kiếm trên danh sách để tìm trong một tập hợp một phần tử chứa một khoá nào đó.
- Tìm kiếm tuyến tính là hoạt động tìm kiếm liên tiếp được diễn ra qua tất cả các phần tử.
- Tìm kiếm nhị phân được thực hiện trên mảng đã được sắp xếp, bằng cách so sánh phần tử cần tìm với phần tử tại ví trí giữa nhất của tập dữ liệu.
- Để đánh giá hiệu quả của một thuật toán, có thể xét số các phép tính phải thực hiện khi thực hiện thuật toán
- Thông thường số các phép tính được thực hiện phụ thuộc vào cỡ của bài toán, tức là độ lớn của đầu vào.