Cấu trúc dữ liệu và giải thuật

CÁC CHIẾN LƯỢC TÌM KIẾM

Giảng viên:
Văn Chí Nam – Nguyễn Thị Hồng Nhung – Đặng Nguyễn Đức Tiến – Vũ Thanh Hưng

Nội dung trình bày

- Giới thiệu
- Tìm kiếm tuần tự
- Tìm kiếm nhị phân
- Tìm kiếm theo bảng băm
- Tổng kết

Giới thiệu

- Thao tác tìm kiếm rất phổ biến trong cuộc sống hàng ngày.
  - Tìm kiếm hồ sơ, tập tin.
  - Tìm kiếm tên người trong danh sách.
  - Tìm kiếm địa chỉ của một nơi nào đó (địa chỉ trường Tự nhiên, trung tâm hỗ trợ sinh viên v.v…)

  → Biết đối tượng cần tìm

  - Tìm kiếm phòng trọ .
  - Tìm kiếm những địa điểm du lịch ở Cần Thơ

  → Chỉ biết tính chất đối tượng cần tìm.

Thuật toán tìm kiếm (khi biết đối tượng)

- Có nhiều loại:
  - Tìm kiếm tuần tự (Sequential/ Linear Search)
  - Tìm kiếm nhị phân (Binary Search)
  - …
- Mục tiêu:
  - Tìm hiểu về 2 thuật toán tìm kiếm cơ bản.
  - Phân tích thuật toán để lựa chọn thuật toán phù hợp khi áp dụng vào thực tế.

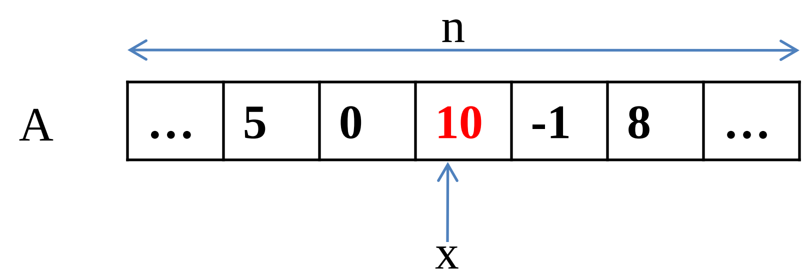
Tìm kiếm tuần tự

Sequential Search

Linear Search

Thuật toán tìm kiếm tuần tự

- Input:
  - Dãy A, *n* phần tử
  - Giá trị *x* cần tìm



- Output:
  - Nếu *x* xuất hiện trong A: trả về vị trí xuất hiện đầu tiên của *x*
  - Nếu không: trả về n hoặc -1
- Thuật toán:
  - Vét cạn (exhaustive)
  - Dùng lính canh (sentinel)

Tìm kiếm tuần tự - Vét cạn

- Thuật toán:
  - Lần lượt so sánh x với các phần tử của mảng A cho đến khi gặp được phần tử cần tìm, hoặc hết mảng.
  - Ví dụ: A = {1, 25, 6, 5, 2, 37, 40}, x = 6

x = 6

1	25	6	5	2	37	40

x = 6

1	25	6	5	2	37	40

x = 6

1	25	6	5	2	37	40

Dừng

Tìm kiếm tuần tự - Vét cạn

Thuật toán: **LinearExhaustive**

- Bước 1. Khởi tạo biến chỉ số: *i = 0*
- Bước 2. Kiểm tra xem có thực hiện hết mảng hay chưa: *So sánh i và n*
  - Nếu chưa hết mảng (i < n), sang bước 3.
  - Nếu đã hết mảng (i >= n), thông báo không tìm thấy giá trị x cần tìm.
- Bước 3. *So sánh giá trị a[i] với giá trị x* cần tìm
  - Nếu a[i] bằng x: Kết thúc chương trình và thông báo đã tìm thấy x.
  - Nếu a[i] khác x, *tăng i thêm 1* và quay lại bước 2.

Code

- Viết hàm linearSearch để tìm một số nguyên x cho trước trong mảng a có n phần tử

```
int linearSearch(int a[], int n, int x)
{
}
```

Hàm trả về vị trí của x trong a nếu tìm thấy

Trong trường hợp không tìm thấy, hàm trả về -1

Tìm kiếm tuần tự - Vét cạn

- Nhận xét: Phép so sánh là phép toán sơ cấp được dùng trong thuật toán. Suy ra, số lượng các phép so sánh sẽ là thước đo độ phức tạp của thuật toán.

- Mỗi vòng lặp có 2 điều kiện cần kiểm tra:
  - Kiểm tra cuối mảng (bước 2)
  - Kiểm tra phần tử hiện tại có bằng *x*? (bước 3)

Bài tập:

- Đếm số phép so sánh của thuật toán tìm kiếm vet cạn trong trường hợp:
  - Tốt nhất
  - Xấu nhất
  - Tại vị trí k bất kì
- Tính độ phức tạp thuật toán trong trường hợp
  - Tốt nhất
  - Xấu nhất
  - Trung bình

Tìm kiếm tuần tự - Vét cạn

- Trường hợp x nằm ở 2 biên của mảng A: rất hiếm khi xuất hiện.
- Ước lượng số vòng lặp trung bình sẽ hữu ích hơn.
- Số phép so sánh trung bình:

  $(2(0 + 1+2+ \ldots + n) + 2n+1)/(n+1) = n+2-1/(n+1)$

=> Số phép so sánh tăng/giảm tuyến tính theo số phần tử

Tìm kiếm tuần tự - Vét cạn

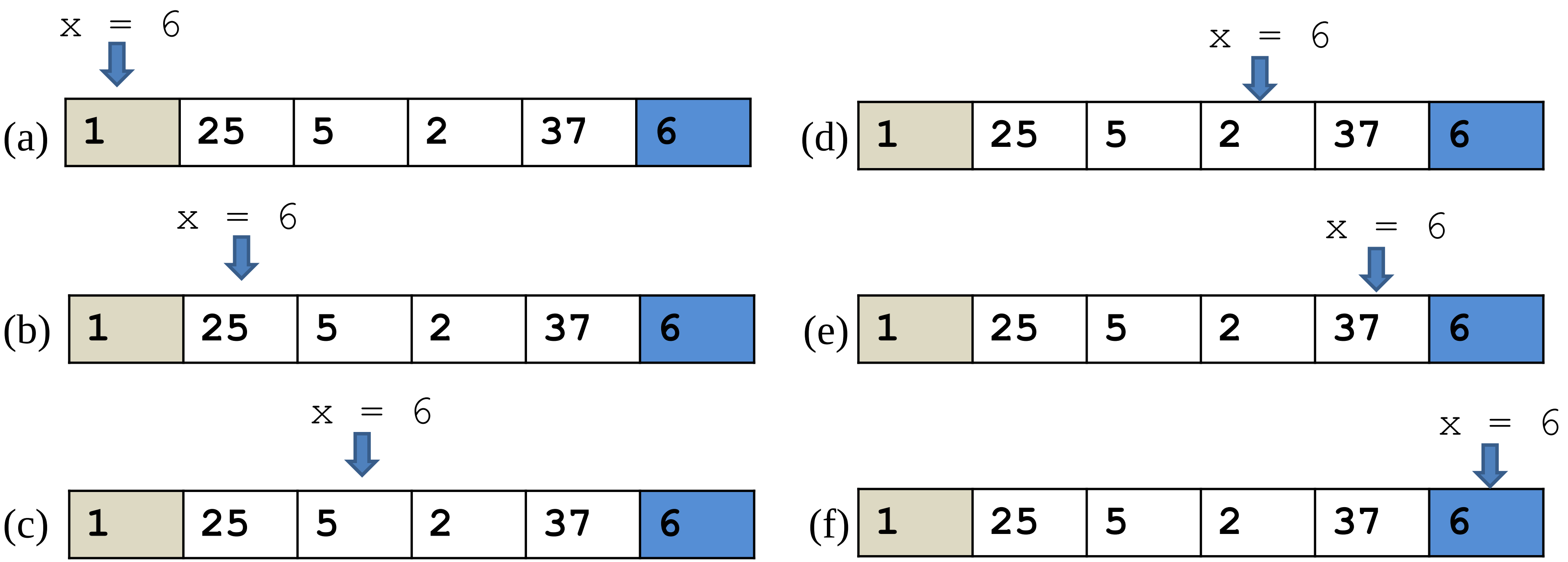
- Vậy độ phức tạp của thuật toán là:
  - Tốt nhất: O(1).
  - Trung bình: O(n).
  - Xấu nhất: O(n).

Tìm kiếm tuần tự - Lính canh

- Trong thuật toán vét cạn, có 2 điều kiện được kiểm tra.
- Có thể bỏ việc kiểm tra điều kiện cuối mảng bằng cách dùng "lính canh".
- **Lính canh là phần tử có giá trị bằng với phần tử cần tìm và đặt ở cuối mảng.**

Tìm kiếm tuần tự - Lính canh

- Ví dụ: A = {1, 25, 5, 2, 37}, x = 6



return 5;

Tìm kiếm tuần tự - Lính canh

Thuật toán: **LinearSentinel**

- Bước 1. Khởi tạo biến chỉ số: *i = 0*
- Bước 2. *So sánh giá trị a[i] với giá trị x* cần tìm
  - Nếu a[i] bằng x:
    - Nếu i < n: Kết thúc chương trình và thông báo đã tìm thấy x.
    - Nếu i >= n: Thông báo không tìm thấy x trong mảng.
  - Nếu a[i] khác x, *tăng i thêm 1* và quay lại bước 2.

Code

- Viết hàm sentinelSearch để tìm một số nguyên x cho trước trong mảng a có n phần tử

int sentinelSearch(int a[], int n, int x)

{

}

Hàm trả về vị trí của x trong a nếu tìm thấy

Trong trường hợp không tìm thấy, hàm trả về -1

Tìm kiếm tuần tự - Lính canh

- Thực nghiệm cho thấy trong trường hợp n lớn, thời gian tìm kiếm giảm khi dùng phương pháp lính canh.
  - Với n =15000: nhanh hơn khoảng 20% (0.22s so với 0.28s)

Tìm kiếm nhị phân

Binary Search

Lớp CK1

1	1161141	Tình
2	1161037	Hà
3	1161107	Quang
4	1161065	Khánh
5	1161010	Bảo
6	1161176	Tiên
7	1161020	Đại
8	1161044	Hồ
9	1161063	Khải
10	1161032	Dung
11	1161102	Phúc
12	1161089	Nghi
13	1161173	Nhân
14	1161147	Triết
15	1161023	Đăng
16	1161011	Bình
17	1161164	Vinh
18	1161086	Nam
19	1161047	Hòa
20	1161123	Thái
21	1161116	Tâm
22	1161052	Hoàng
23	1161171	Tú
24	1161015	Châu
25	1161033	Gon
26	1161055	Hùng
27	1161002	An
28	1161028	Định
29	1161149	Trường
30	1161156	Tùng
31	1161034	Duy
32	1161159	Tuyên
33	1161071	Kiện
34	1161009	Bách
35	1161098	Phong
36	1161079	Lực
37	1161088	Năng
38	1161125	Thành
39	1161174	Thám
40	1161145	Trang
41	1161012	Bình
42	1161008	Trâm Anh
43	1161040	Hiền
44	1161154	Tự
45	1161096	Nhiên
46	1161061	Huy
47	1161167	Vũ
48	1161150	Trọng
49	1161169	Vương
50	1161031	Đức

Thuật toán tìm kiếm nhị phân

- Với dãy A được sắp xếp thứ tự (ví dụ: tăng dần), độ phức tạp của thuật toán tìm kiếm tuần tự không đổi.

- Tận dụng thông tin của mảng đã được sắp xếp để giới hạn vị trí của giá trị cần tìm trong mảng.

-> Thuật toán tìm kiếm nhị phân.

Thuật toán tìm kiếm nhị phân

- Input:
  - Dãy A, *n* phần tử **đã được sắp xếp**
  - Giá trị *x* cần tìm
- Output:
  - Nếu *x* xuất hiện trong A: trả về một vị trí xuất hiện của *x*
  - Nếu không: trả về n hoặc -1

Thuật toán tìm kiếm nhị phân

- Ý tưởng:
  - So sánh x với phần tử chính giữa mảng A.
    - Nếu x là phần tử giữa thì dừng.
  - Nếu không: xác định xem ***x*** có thể thuộc nửa trái hay nửa phải của A.
  - Lặp lại 2 bước trên với nửa đã được xác định.

Thuật toán tìm kiếm nhị phân

Thuật toán: BinarySearch(A[], n, x)

- Bước 1. Khởi gán left = 0 và right = n – 1.
- Bước 2. Trong khi left <= right, thực hiện:
  - 2.1. Đặt *mid = (left + right)/2*
  - 2.2. *So sánh giá trị x và a[mid]*:
    - Nếu x < a[mid], gán *right = mid – 1*.
    - Nếu x > a[mid], gán *left = mid + 1*.
    - Nếu x = a[mid], thông báo đã tìm thấy x và kết thúc.
- Kết quả trả về không tìm thấy x nếu left > right*.

** Điều này có nghĩa là không còn phần tử nào trong mảng: x không có trong mảng*

Thuật toán tìm kiếm nhị phân

Cài đặt đệ quy: BinarySearch(A[], int n, left, right, x)

- Bước 1. Nếu left > right: thông báo không tìm thấy x và thoát khỏi hàm.
- Bước 2.
  - 2.1. Đặt *mid = (left + right)/2*
  - 2.2. *So sánh giá trị x và a[mid]*:
    - Nếu x < a[mid], Gọi BinarySearch(A, left, mid – 1, x)
    - Nếu x > a[mid], Gọi BinarySearch(A, mid + 1, right, x)
    - Nếu x = a[mid], thông báo đã tìm thấy x và kết thúc (trả lại giá trị mid).

Code

- Cài đặt hàm binarySearch

```
int binarySearch(int a[], int n, int left, int right, int x)
{
}
```

Trả về vị trí x trong a nếu tìm thấy, ngược lại trả về -1.

Cài đặt hàm binarySearch (không đệ quy)

Int binarySearch(int a[], int n, int left, int right, int x)

{

- Khởi tạo left = 0;
- Khởi tạo right = n-1;
- Trong khi (left <=right)

  {

  - 2.1. Đặt *mid = (left + right)/2*
  - 2.2. *So sánh giá trị x và a[mid]*:
    - Nếu x < a[mid], gán *right = mid – 1*.
    - Nếu x > a[mid], gán *left = mid + 1*
    - Nếu x = a[mid], thông báo đã tìm thấy x và kết thúc.

  }

- return ?????; // left > right

}

Thuật toán tìm kiếm nhị phân

- Minh họa:
  - A[] = {1, 2, 6, 26, 28, 37, 40}, x = 2

index	0	1	2	3	4	5	6
A[i]	1	2	6	26	28	37	40
Vòng 1	**left**			**mid**			**right**
Vòng 2	**left**	**mid**	**right**				

x = a[1] -> **return** 1

Thuật toán tìm kiếm nhị phân

- Minh họa:
  - A[] = {1, 2, 6, 26, 28, 37, 40}, x = 40

index	0	1	2	3	4	5	6
A[i]	1	2	6	26	28	37	40
Vòng 1	**left**			**mid**			**right**
Vòng 2					**left**	**mid**	**right**
Vòng 3							**left** **mid** **right**

x = a[6] -> **return** 6

Thuật toán tìm kiếm nhị phân

- Minh họa:
  - A[] = {1, 2, 6, 26, 28, 37, 40}, x = -7

index	0	1	2	3	4	5	6
A[i]	1	2	6	26	28	37	40
Vòng 1	**left**			**mid**			**right**
Vòng 2	**left**	**mid**	**right**				
Vòng 3	**left** **mid** **right**						
Vòng 4							

right = -1, left = 0

=> **right < left** => thoát khỏi while,

return -1

Thuật toán tìm kiếm nhị phân

- Phân tích thuật toán tuyến tính:
  - Mỗi lần lặp thì chiều dài của mảng con giảm *khoảng* ½ so với mảng trước đó.
  - $n = 2^k + m \ (0 \leq m < 2)$
  - $2^k \leq n < 2^{k+1} \Rightarrow k \leq \log_2 n < k+1 \Rightarrow k = \lfloor \log_2 n \rfloor$

  => mảng A ban đầu được chia nửa *khoảng* **k** lần.
  - Số lần thực hiện vòng while là khoảng k lần, mỗi vòng lặp thực hiện 1 phép so sánh.

Thuật toán tìm kiếm nhị phân

- Phân tích thuật toán tuyến tính:
  - Trường hợp tốt nhất: k = 1 ⇔ x là phần tử chính giữa của mảng.
  - Trường hợp xấu nhất: $k = \lfloor \log_2 n \rfloor + 1$ ⇔ x không thuộc mảng hoặc x là phần tử cuối cùng của mảng

=> Số phép so sánh tăng theo hàm logarit

Thuật toán tìm kiếm nhị phân

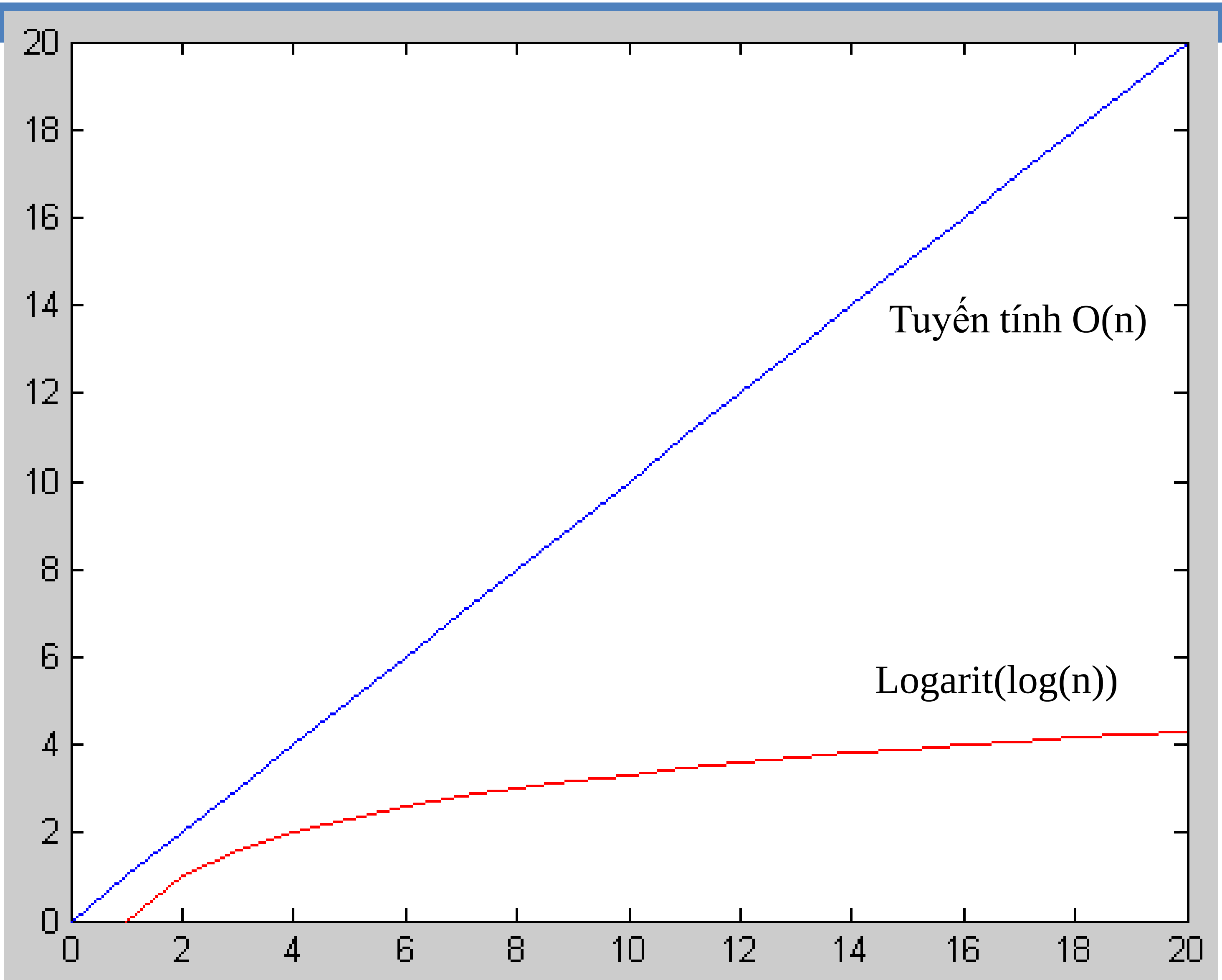
- Độ phức tạp của tìm kiếm nhị phân
  - Trường hợp tốt nhất: O(1)
  - Trường hợp trung bình: $O(\log_2 n)$
  - Trường hợp xấu nhất: $O(\log_2 n)$

So sánh hiệu suất

- So sánh trường hợp xấu nhất của 2 thuật toán:

Kích thước mảng	T/h xấu nhất	
	Tuần tự	Nhị phân
100.000	100.000	16
200.000	200.000	17
400.000	400.000	18
800.000	800.000	19
1.600.000	1.600.000	20

O(log(n)) vs O(n)



Tổng kết

- Có nhiều thuật toán tìm kiếm, ước lượng số phép so sánh của mỗi thuật toán cho biết hiệu suất của thuật toán.
- Thuật toán tuần tự tìm kiếm cho đến khi tìm thấy giá trị cần tìm hoặc hết mảng
- Hiệu suất của tìm kiếm tuần tự trong trường hợp xấu nhất là 1 hàm tuyến tính theo số phần tử mảng.

Tổng kết

- Nếu mảng đã được sắp xếp thì nên dùng tìm kiếm nhị phân.
- Tìm kiếm nhị phân dùng kết quả của phép so sánh để thu hẹp vùng tìm kiếm kế tiếp.
- Hiệu suất của tìm kiếm nhị phân là một hàm logarit theo số phần tử mảng.

Hỏi và Đáp