ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KHOA HỌC MÁY TÍNH

GIẢI BÀI TẬP

PHÂN TÍCH ĐỘ PHỨC TẠP THUẬT TOÁN KHÔNG ĐỆ QUY

Môn học: Phân tích và thiết kế thuật toán

(CS112.N21.KHTN)

Sinh viên thực hiện: Hà Văn Hoàng - 21520033

Võ Thị Phương Anh - 21522883

Thành phố Hồ Chí Minh, ngày 18 tháng 3 năm 2022

Mục lục

1. Bài tập 1 (Bài 10, trang 60 sách giáo trình):

The range of a finite nonempty set of n real numbers S is defined as the difference between the largest and smallest elements of S. For each representation of S given below, describe in English an algorithm to compute the range. Indicate the time efficiency classes of these algorithms using the most appropriate notation.

a. An unsorted array

b. A sorted array

c. A sorted singly linked list

d. A binary search tree

Phạm vi của một tập hợp hữu hạn n số thực S khác rỗng được xác định bằng khoảng cách giữa phần tử lớn nhất và phần tử nhỏ nhất của S. Đối với mỗi dạng biểu diễn của S được đưa ra dưới đây, hãy mô tả một thuật toán để tính phạm vi đó. Chỉ ra các lớp hiệu suất thời gian của thuật toán bằng ký hiệu phù hợp nhất.

a. Một mảng chưa được sắp xếp

b. Một mảng đã được sắp xếp

c. Một danh sách liên kết đơn đã được sắp xếp

d. Một cây tìm kiếm nhị phân

1.1. Giải

1.1.1. Câu a

Đối với một mảng chưa được sắp xếp, các phần tử có thể nằm xen lẫn nhau và không tuân theo một thứ tự nào. Vì thế ta sẽ tìm các phần tử nhỏ nhất và lớn nhất trong mảng bằng cách duyệt qua mảng một lần.

Cách thực hiện: Khởi tạo hai biến cho phần tử đầu tiên của mảng và so

sánh từng phần tử tiếp theo với các biến này. Nếu tìm thấy một phần tử nhỏ hơn phần tử nhỏ nhất hiện tại, chúng ta cập nhật biến nhỏ nhất. Nếu tìm thấy một phần tử lớn hơn phần tử lớn nhất hiện tại, chúng ta cập nhật biến lớn nhất. Cuối cùng, sau khi tìm được phần tử nhỏ nhất và lớn nhất, chúng ta tính toán khoảng cách của tập hợp bằng cách trừ phần tử nhỏ nhất từ phần tử lớn nhất. Minh họa thuật toán bằng Python:

```python
import array as arr

min_element = arr[0]
max_element = arr[0]

for i in range(1, len(arr)):
    if arr[i] < min_element:
        min_element = arr[i]
    elif arr[i] > max_element:
        max_element = arr[i]

range_of_arr = max_element - min_element
```

Độ phức tạp thời gian của thuật toán này là O(n), vì chúng ta cần duyệt qua toàn bộ mảng một lần.

1.1.2. Câu b

Đối với mảng đã được sắp xếp, ta có thể dễ dàng xác định phần tử lớn nhất trong mảng là phần tử cuối cùng và phần tử nhỏ nhất là phần tử đầu tiên. Để tính phạm vi của tập hợp, thuật toán chỉ cần thực hiện trừ 2 phần tử trên với nhau. Do đó độ phức tạp thời gian của thuật toán là O(1).

1.1.3. Câu c

Đối với một danh sách liên kết đơn đã được sắp xếp, chúng ta có thể tìm các phần tử nhỏ nhất và lớn nhất bằng cách duyệt qua list này một lần, tương

tự như trong thuật toán mảng chưa được sắp xếp. Giữ lại các phần tử nhỏ nhất và lớn nhất đã thấy cho đến khi duyệt qua toàn bộ danh sách và tính phạm vi bằng cách trừ phần tử nhỏ nhất từ phần tử lớn nhất.

Mô tả thuật toán bằng Python:

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def find_range(head):
    min_val = head.data
    max_val = head.data

    current = head
    while current:
        if current.data < min_val:
            min_val = current.data
        elif current.data > max_val:
            max_val = current.data
        current = current.next

    return max_val - min_val
```

Độ phức tạp thời gian của thuật toán này là O(n), vì chúng ta cần duyệt qua toàn bộ danh sách một lần.

1.1.4. Câu d

Đối với cây nhị phân tìm kiếm, các phần tử nhỏ nhất và lớn nhất có thể được tìm bằng cách lần lượt duyệt qua các nút ngoài cùng bên trái và ngoài cùng bên phải của cây. Bắt đầu từ nút gốc và đi qua trái cho đến khi đến nút lá, nút lá này sẽ chứa phần tử nhỏ nhất trong cây. Tương tự, bắt đầu từ nút gốc và đi qua phải cho đến khi đến nút lá, nút này sẽ chứa phần tử lớn nhất trong cây. Cuối cùng, tính toán phạm vi bằng cách lấy số lớn nhất trừ đi số nhỏ nhất.

Chương trình mô tả thuật toán bằng Python:

```python
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

def find_min(root):
    current = root

    while(current.left is not None):
        current = current.left

    return current.data

def find_max(root):
    current = root

    while(current.right is not None):
        current = current.right

    return current.data

def find_range(root):
    if root is None:
        return None

    min_val = find_min(root)
    max_val = find_max(root)

    return max_val - min_val
```

Độ phức tạp về thời gian trung bình của thuật toán này là $\Theta(\log n)$ trong trường hợp chỉ cần đi qua một phần của cây và là $O(n)$ trong trường hợp xấu nhất nếu cây không cân bằng.

2. Bài tập 2 (Bài 11, trang 61 sách giáo trình):

Lighter or heavier? You have $n > 2$ identical-looking coins and a two-pan balance scale with no weights. One of the coins is a fake, but you do not know whether it is lighter or heavier than the genuine coins, which all weigh the same. Design a Omega(1) algorithm to determine whether the fake coin is lighter or heavier than the others.

Nhẹ hơn hay nặng hơn? Cho $n > 2$ đồng xu trông giống nhau và một cái

cân cân bằng với 2 bên cân. Trong các đồng xu có 1 đồng xu là giả, nhưng ta không biết nó nhẹ hơn hay nặng hơn so với các đồng xu thật, tất cả đều có cùng trọng lượng. Thiết kế một thuật toán có độ phức tạp Omega(1) để xác định liệu đồng xu giả nhẹ hơn hay nặng hơn so với các đồng xu khác. **2.1. Giải**

- Ý tưởng: Chia để trị.
  - Chia các đồng xu thành ba nhóm gồm n/3 đồng xu mỗi nhóm và cân hai trong số các nhóm này với nhau. Nếu 2 nhóm cân bằng thì đồng giả phải nằm trong n/3 nhóm còn lại. Mặt khác, chúng ta biết rằng đồng xu giả nằm trong nhóm nặng hơn (nếu nó nhẹ hơn) hoặc trong nhóm nặng hơn (nếu nó nặng hơn).
  - Lặp lại với nhóm gồm n/3 đồng xu, chia nó thành ba nhóm mỗi nhóm gồm n/9 đồng xu và cân hai trong số các nhóm này với nhau. Một lần nữa, nếu hai nhóm cân bằng, thì đồng xu giả phải nằm trong nhóm n/9 đồng xu còn lại. Mặt khác, chúng ta biết rằng đồng xu giả nằm trong nhóm nặng hơn (nếu nó nhẹ hơn) hoặc trong nhóm nặng hơn (nếu nó nặng hơn).
  - Tiếp tục quá trình này cho đến khi chỉ còn lại hai đồng xu, tại thời điểm đó, chúng ta có thể chỉ cần cân chúng với nhau để xác định đồng nào là giả và đồng nào nhẹ hơn hay nặng hơn.
  - Để thuật toán này hoạt động hiệu quả (độ phức tạp O(1)), ta sẽ luôn chia số lượng đồng xu cho hệ số 3, vì vậy số lần cân cần thiết là O(log n) (trong cơ số 3), là một hằng số. Do đó, độ phức tạp thời gian của thuật toán là O(1).
- Mô tả thuật toán bằng Python (mã giả):

```python
def find_fake_coin(coins):
    n = len(coins)
    group_size = n // 3

    # Cân lần đầu tiên
    left = coins[:group_size]
    middle = coins[group_size:2*group_size]
    right = coins[2*group_size:3*group_size]

    result = compare(left, middle)  # compare() là một hàm so sánh hai tập đồng xu trên cân bằng
    if result == 0:
        # Đồng xu giả nằm trong một trong hai nhóm còn lại
        if n % 3 == 1:
            return coins[-1]  # Chỉ còn một đồng xu, đó chính là đồng giả
        elif n % 3 == 2:
            # Cân hai đồng xu còn lại để xác định đồng giả
            result = compare([coins[-2]], [coins[-1]])
            if result == -1:
                return coins[-1]  # Đồng xu giả nặng hơn
            else:
                return coins[-2]  # Đồng xu giả nhẹ hơn
        else:
            coins = right  # Đồng xu giả nằm trong nhóm phía bên phải
    elif result == -1:
        coins = left  # Đồng xu giả nằm trong nhóm bên trái và nhẹ hơn
    else:
        coins = middle  # Đồng xu giả nằm trong nhóm giữa và nặng hơn
```

```python
    # Cân các nhóm tiếp theo
    while len(coins) > 2:
        group_size = len(coins) // 3
        left = coins[:group_size]
        middle = coins[group_size:2*group_size]
        right = coins[2*group_size:3*group_size]
        result = compare(left, middle)
        if result == 0:
            coins = right
        elif result == -1:
            coins = left
        else:
            coins = middle

    # Cân cuối cùng
    result = compare([coins[0]], [coins[1]])
    if result == -1:
        return coins[1]  # Đồng xu giả nặng hơn
    else:
        return coins[0]  # Đồng xu giả nhẹ hơn
```

3. Bài tập 3 (Bài 11, trang 69 sách giáo trình):

Consider the following version of an important algorithm that we will study later in the book.

ALGORITHM $GE(A[0..n-1, 0..n])$

//Input: An $n \times (n+1)$ matrix $A[0..n-1, 0..n]$ of real numbers

for $i \leftarrow 0$ **to** $n-2$ **do**

  for $j \leftarrow i+1$ **to** $n-1$ **do**

    for $k \leftarrow i$ **to** n **do**

      $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$

a. Find the time efficiency class of this algorithm.

b. What glaring inefficiency does this pseudo-code contain and how can it be eliminated to speed the algorithm up?

Cho giải thuật bên dưới:

ALGORITHM $GE(A[0..n-1, 0..n])$
//Input: An $n \times (n+1)$ matrix $A[0..n-1, 0..n]$ of real numbers
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    for $k \leftarrow i$ **to** n **do**
      $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$

a. Tìm độ phức tạp thời gian của thuật toán này.

b. Thuật toán còn điểm nào chưa tối ưu và làm cách nào để chỉnh sửa cho thuật toán nhanh hơn?

3.1. Giải

3.1.1. Câu a

Trong thuật toán bên, vòng lặp ngoài cùng lặp lại n-1 lần, vòng lặp thứ hai lặp lại (n-1)+(n-2)+...+1 = (n-1)n/2 lần, và vòng lặp trong cùng lặp lại n lần. Bên trong vòng lặp trong cùng, có hai phép trừ, một phép nhân và một phép chia. Do đó, tổng số thao tác là khoảng $(n-1)(n^2)/2$, nằm trong lớp hiệu suất thời gian (độ phức tạp thời gian) là $O(n^3)$.

3.1.2. Câu b

Điểm chưa tối ưu trong thuật toán ở câu a là nó thực hiện cùng một phép chia (A[j, i] / A[i, i]) nhiều lần trong vòng lặp trong cùng cho cùng một giá trị i. Điều này có thể được giảm thiểu bằng cách tính giá trị của A[i, i] bên ngoài vòng lặp và lưu trữ nó trong một biến tạm thời.

Sau đó, chúng ta có thể sử dụng giá trị này để tính toán tất cả các phép

chia cần thiết trong vòng lặp trong. Điều này giảm số lần thực hiện phép chia từ n^3 xuống n^2, giúp tăng tốc độ thuật toán. Mã giả được sửa đổi sẽ là:

ALGORITHM GE(A[0..n − 1, 0..n])
//Input: An n × (n + 1) matrix A[0..n − 1, 0..n] of real numbers
for i ← 0 to n − 2 do
pivot ← A[i, i]
for j ← i + 1 to n − 1 do
factor ← A[j, i] / pivot
for k ← i to n do
A[j, k] ← A[j, k] − A[i, k] ∗ factor

Với sự thay đổi này, độ phức tạp về thời gian của thuật toán vẫn ở mức $O(n^3)$, nhưng thời gian chạy thực tế nhanh hơn đáng kể do giảm các thao tác chia.