Text, letter

Description automatically generated

a. To compute the range of a nonempty set S with unsorted array, we can first find the minimum and maximum elements of the array by scanning through it once. We initialize the minimum value and maximum value to the first element of the array, and then compare the current element with the minimum and maximum values as we traverse through the array. If the current element is smaller than the minimum value, we update the minimum value. Similarly, if the current element is larger than the maximum value, we update the maximum value. Once we reach the end of the array, we can calculate the range as the difference between the maximum and minimum values.

Time complexity:

Worst-case: O(n) - when we need to traverse the full length of the array to find the minimum or maximum element..

Best-case: Ω(n) - same as worst-case.

Average-case: Θ(n) - same as worst-case.

b. If the array is sorted, we can simply subtract the first element from the last element to obtain the range.

Time complexity:

Worst-case: O(1) - since we just need to perform a subtraction operation without iterating.

Best-case: Ω(1) - same as worst-case.

Average-case: Θ(1) - same as worst-case.

c. Similar to unsorted array, we need to traverse the list to find its minimum and maximum elements.

Time complexity:

Worst-case: O(n) - when we need to traverse the full length of the linked list to find the minimum or maximum element.

Best-case: Ω(n) - same as worst-case.

Average-case: Θ(n) - same as worst-case.

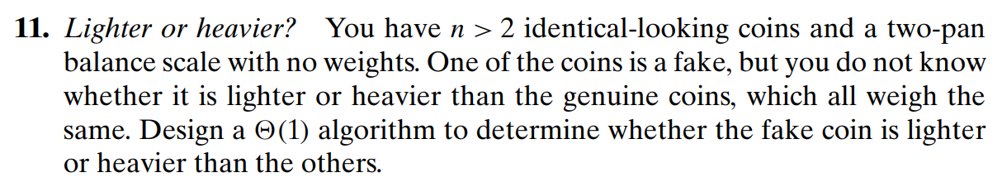
d. With a binary search tree, we can find the minimum and maximum elements by traversing down the the left and right subtrees, respectively, until we reach the leaves. Once we have found the minimum and maximum elements, we can compute the range as the difference between them.

Time complexity:

Worst-case: O(n) - when the binary search tree is unbalanced, and we need to traverse the full length of the tree to find the minimum or maximum element.

Best-case: Ω(log n) - when the binary search tree is balanced and the minimum and maximum elements are located at opposite ends of the tree, we can find them in logarithmic time.

Average-case: Θ(log n) - when the binary search tree is balanced.



Chia thành 3 chồng bằng nhau:

- Nếu n chia hết cho 3 => Qua 2 lần cân, chắc chắn có 2 chồng bằng nhau và chồng còn lại khác khối lượng => biết nặng hơn hay nhẹ hơn.

- Nếu n chia 3 dư 1 => Qua 2 lần cân:

+ Nếu 3 chồng bằng nhau => đồng dư ra là giả => lấy 1 đồng bất kỳ trong 3 chồng cân với đồng dư sẽ biết.

+ Ngược lại, chắc chắn có 2 chồng bằng nhau và chồng còn lại khác khối lượng => biết nặng hơn hay nhẹ hơn.

- Nếu n chia 3 dư 2:

+ Nếu 3 chồng bằng nhau => đồng giả nằm trong 1 trong 2 đồng dư ra => trong 2 chồng đầu, thay thế đồng xu đầu tiên bằng đồng xu dư ra => Cân lần lượt 2 chồng đó với chồng cuối sẽ biết được.

+ Ngược lại, qua 2 lần cân chắc chắn có 2 chồng bằng nhau và chồng còn lại khác khối lượng => biết nặng hơn hay nhẹ hơn.

Vậy để biết được đồng xu giả ta chỉ qua hữu hạn các bước mà không phụ thuộc vào n => Θ(1).

Text

Description automatically generated

a. Vòng lặp ngoài cùng chạy (n-2) lần, vòng lặp ở trong chạy (n-i-1) lần, vòng lặp trong cùng chạy (n-i) lần => O(n3).

b.

Xét *i* = 0, *j* € [*i*+1, *n*-1], *k* = *i*, ta có:

a[1, 0] = a[1, 0] - a[0, 0] \* a[1, 0] / a[0, 0]

a[2, 0] = a[2, 0] - a[0, 0] \* a[2, 0] / a[0, 0]

…

a[*n*-1, 0] = a[*n*-1, 0] - a[0, 0] \* a[*n*-1, 0] / a[0, 0]

→ a[*j,* 0] = a[*j*, 0] – a[0, 0] \* a[*j*, 0] / a[0, 0] = a[*j*, 0] - a[*j*, 0] = 0

VD: *(\* là giữ nguyên giá trị ban đầu, bỏ trống là chưa biết giá trị)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | … | *n - 2* | *n - 1* | *n* |
| 0 | \* | \* | \* | \* | \* | \* | \* |
| 1 | 0 |  |  |  |  |  |  |
| 2 | 0 |  |  |  |  |  |  |
| … | 0 |  |  |  |  |  |  |
| *n - 2* | 0 |  |  |  |  |  |  |
| *n - 1* | 0 |  |  |  |  |  |  |

Tổng quát lên với mọi *i* € [*0*, *n*-2] và *j* € [*i*+1, *n*-1]*,* ta có: a[*j, i*] = 0

VD: *(\* là giữ nguyên giá trị ban đầu, bỏ trống là chưa biết giá trị)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | … | *n - 2* | *n - 1* | *n* |
| 0 | \* | \* | \* | \* | \* | \* | \* |
| 1 | 0 |  |  |  |  |  |  |
| 2 | 0 | 0 |  |  |  |  |  |
| … | 0 | 0 | 0 |  |  |  |  |
| *n - 2* | 0 | 0 | 0 | 0 |  |  |  |
| *n - 1* | 0 | 0 | 0 | 0 | 0 |  |  |

Xét *i* = 0, *j* = *i* + 1 = 1 và *k* € [*i+1*, *n*], ta có:

a[1, 1] = a[1, 1] - a[0, 1] \* a[1, 0] / a[0, 0]

a[1, 2] = a[1, 2] - a[0, 2] \* a[1, 0] / a[0, 0]

…

a[1, *n*] = a[1, *n*] - a[0, *n*] \* a[1, 0] / a[0, 0]

→ a[1, *k*] = a[1, *k*] - a[0, *k*] \* a[1, 0] / a[0, 0] = a[1, *k*] - a[0, *k*] \* 0 / a[0, 0] = a[1, *k*]

→ Giá trị a[1, *k*] giữ nguyên.

VD: *(\* là giữ nguyên giá trị ban đầu, bỏ trống là chưa biết giá trị)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | … | *n - 2* | *n - 1* | *n* |
| 0 | \* | \* | \* | \* | \* | \* | \* |
| 1 | 0 | \* | \* | \* | \* | \* | \* |
| 2 | 0 | 0 |  |  |  |  |  |
| … | 0 | 0 | 0 |  |  |  |  |
| *n - 2* | 0 | 0 | 0 | 0 |  |  |  |
| *n - 1* | 0 | 0 | 0 | 0 | 0 |  |  |

Tổng quát lên với mọi *j* € [1, *n-1*], *k* € [*j*, *n*] thì giá trị a[*j*, *k*] giữ nguyên.

VD: *(\* là giữ nguyên giá trị ban đầu)*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | … | *n - 2* | *n - 1* | *n* |
| 0 | \* | \* | \* | \* | \* | \* | \* |
| 1 | 0 | \* | \* | \* | \* | \* | \* |
| 2 | 0 | 0 | \* | \* | \* | \* | \* |
| … | 0 | 0 | 0 | \* | \* | \* | \* |
| *n - 2* | 0 | 0 | 0 | 0 | \* | \* | \* |
| *n - 1* | 0 | 0 | 0 | 0 | 0 | \* | \* |

Vậy ta chỉ cần 2 vòng lặp để gán giá trị 0 cho các ô như trên nên độ phức tạp thời gian là O(n2).