Bài tập thực hành Tuần 1
Môn: Phương pháp Toán cho Trí tuệ nhân tạo

Lê Nguyễn - 21120511

Ngày 11 tháng 3 năm 2023

Bài 1:

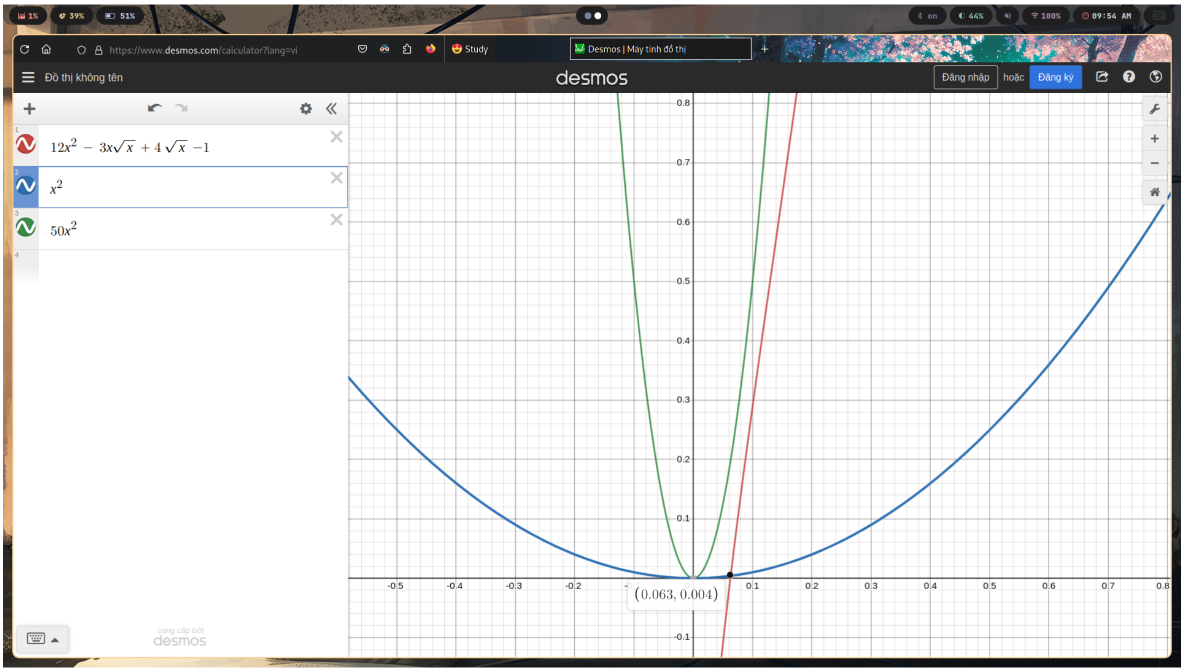
(a) Ta có:

$$g(n) = (3n + \frac{1}{\sqrt{n}})(4n - \sqrt{n}) = 12n^2 - 3n\sqrt{n} + 4\sqrt{n} - 1$$
$$= 12n^2 + \sqrt{n}(4 - 3n) - 1$$

Ta thấy:

$$n^2 \leq 12n^2 + \sqrt{n}(4 - 3n) - 1 \leq 50n^2, \quad \forall x \geq 0.063$$

Do đó $g(n) \in \Theta(n^2)$.



(b) Xét hàm số $f(n)$ bất kì thuộc $\Omega(n^3)$, ta có:

$$f(n) \geq c_0 n^3, \quad \forall n \geq n_0$$

Khi đó tồn tại cặp số (c_1, n_0) sao cho:

$$f(n) \geq c_0 n^3 \geq c_1 n^2 \quad \forall n \geq n_0$$

Do đó $f(n) \in \Omega(n^2)$. Vậy $\Omega(n^3) \subseteq \Omega(n^2)$.

Bài 2:

(a) Đặt thời gian chạy của cả chương trình là $T(n)$ và t_i là số lần chạy của vòng for lồng phía trong. Ứng với mỗi dòng ta sẽ mất đi c_i tài nguyên với i là số dòng.

Ta có:

- Dòng đầu tiên (bắt đầu từ vòng for đầu tiên) mất $c_1\left(\frac{n}{3}+2\right)$
- Dòng thứ hai mất:

$$c_2 \sum_{i=1}^{n/3+1} t_i = c_2\left(\frac{n}{3}+1\right) t_i \quad \text{(do vòng for thứ hai không phụ thuộc vào } i\text{)}$$

- Dòng thứ ba mất:

$$c_3 \sum_{i=1}^{n/3+1} (t_i - 1) = c_3\left(\frac{n}{3}+1\right)(t_i - 1)$$

Kết hợp cả 3 và ta biết $t_i = \left\lfloor \frac{n}{4}+1 \right\rfloor + 1$, ta được:

$$\begin{aligned} T(n) &= c_1\left(\frac{n}{3}+2\right) + c_2\left(\frac{n}{3}+1\right)\left(\frac{n}{4}+2\right) + c_3\left(\frac{n}{3}+1\right)\left(\frac{n}{4}+1\right) \\ &= \left(\frac{c_2}{12}+\frac{c_3}{12}\right)n^2 + \left(\frac{c_1}{3}+\frac{3c_2}{4}+\frac{5c_3}{12}\right)n + 2c_1 + 2c_2 + c_3 \end{aligned}$$

$\Rightarrow$ Do là một hàm bậc 2 nên thời gian chạy là $\Theta(n^2)$.

(b) Đặt tương tự như câu a, ta có:

- Dòng đầu tiên ta mất $c_1(n+1)$.
- Dòng thứ hai mất:

$$c_2 \sum_{i=1}^{n} t_i$$

- Dòng thứ ba mất:

$$c_3 \sum_{i=1}^{n} (t_i - 1)$$

Dựa theo câu a ta có được $t_i = \left\lfloor \frac{n}{i}+1 \right\rfloor + 1$

Xét 2 tổng ta được:

$$\begin{aligned}\sum_{i=1}^{n} t_j &= \sum_{i=1}^{n}\left(\frac{n}{i}+2\right)\\ &= n+\frac{n}{2}+...+1+2n\\ &= n\left(\sum_{i=1}^{n}\frac{1}{i}\right)+2n\end{aligned}$$

và

$$\begin{aligned}\sum_{i=1}^{n}(t_j-1) &= \sum_{i=1}^{n}\left(\frac{n}{i}+1\right)\\ &= n+\frac{n}{2}+...+1+n\\ &= n\left(\sum_{i=1}^{n}\frac{1}{i}\right)+n\end{aligned}$$

Chuỗi $\sum_{i=1}^{n}\frac{1}{i}$ còn được gọi là số Harmonic và có giá trị xấp xỉ là $\ln(n)$.

Vậy kết hợp lại ta được:

$$c_1(n+1)+c_2(n\ln(n)+2n)+c_3(n\ln(n)+n)$$

$\Rightarrow$ Vậy độ phức tạp của thuật toán trên là $\Omega(n\ln(n))$

> **Note.** Do số Harmonic là xấp xỉ xuống và bỏ các thành phần liên quan nên khi chạy thực tế thì số lần chạy sẽ cao hơn $n\ln(n)$ nên em chọn Ω, đồng thời qua nhiều lần dùng máy tính chạy thử (chạy tới $n = 10000000$) thì em thấy vẫn đúng.

Bài 3:

(a) $O(n^2)$.

- Mục đích của ta là lấy tổng số phần tử của mảng trừ đi số phần tử bị lặp thì sẽ được số phần tử phân biệt có trong mảng.
- Để tìm được số phần tử bị lặp. Ta tạo một biến **count** để lưu số phần tử bị lặp.
- Sau đó ta chạy vòng lặp cho i từ 1 đến $n-1$ và một vòng lặp phía trong cho j chạy từ 0 đến $i-1$. Nếu có phần tử $A[j]$ nào bằng với $A[i]$ thì ta tăng biến **count** lên 1.
- Sau khi kết thúc vòng lặp ta chỉ cần lấy độ dài của mảng trừ cho **count**.

```python
from typing import List

def CountDistinct(A: List[int]) -> int:
    count: int = 0
    n: int = len(A)

    for i in range(1, n):
        for j in range(0, i):
            if (A[j] == A[i])
                count = count + 1

    return n - count
```

- Trong lúc chạy ta không tạo ra mảng phụ nên độ phức tạp không gian là $O(1)$.

(b) $O(n \log n)$.

- Ở cách này, đầu tiên ta sẽ sắp xếp lại các phần tử trong mảng dùng phép sắp xếp có độ phức tạp là $O(n \log n)$ và để tối ưu nhất ta dùng **Quick Sort** (bởi vì **Merge Sort** sẽ tạo ra array phụ trong lúc sắp xếp).
- Sau đó ta lặp mảng A từ đầu đến cuối. Nếu phần tử trước khác phần tử sau thì ta sẽ tăng biến count lên 1.
- Khi đó biến count chính là só phần tử phân biệt trong mảng.

```python
from typing import List

def Partition(A: List[int], low: int, high: int) -> int:
    pivot: int = A[high]
    i: int = low - 1

    for j in range(low, high):
        if (A[j] <= pivot):
            i = i + 1
            (A[i], A[j]) = (A[j], A[i])

    (A[i+1], A[high]) = (A[high], A[i+1])

    return i + 1

def QuickSort(A: List[int], low: int, high: int) -> None:
    if (low < high):
        p = Partition(A, low, high)
        QuickSort(A, low, p - 1)
        QuickSort(A, p + 1, high)

def CountDistinct(A: List[int]) -> int:
    count: int = 0
    n: int = len(A)

    QuickSort(A, 0, n-1)

    for i in range(n-1):
        if (A[i] != A[i+1]):
            count = count + 1

    return count
```

- Trong lúc chạy ta không tạo ra mảng phụ nên độ phức tạp không gian là $O(1)$.

(c) $O(n)$.

- Ở cách này, đầu tiên ta sẽ tạo một array mới (gọi là mảng B) có độ dài bằng với độ lớn của phần tử lớn nhất trong mảng cần xét (gọi là mảng A) cộng thêm 1.
- Sau đó ta lặp qua mảng A thí dụ A có phần tử 7 thì $B[7]$ tăng thêm 1. Ta sẽ làm quy luật như vậy cho đến hết mảng.
- Cuối cùng ta chỉ cần lặp qua mảng B, nếu có phần tử nào lớn hơn 0 thì ta tăng thêm 1 vào biến count. Và cuối cùng thì biến count chính là số phần tử phân biệt trong mảng.
- Ngoài ra để chắc chắn thuật toán chạy đúng, ta lấy mỗi phần tử trong A trừ cho phần tử nhỏ nhất trong A, khi đó trong A sẽ không có phần tử nào bị âm.

```python
from typing import List

def CountDistinct(A: List[int]) -> int:
    count: int = 0
    n: int = len(A)

    minA: int = min(A)

    # tru cac phan tu trong A cho min(A) de khong co phan tu nao < 0
    for k in range(0, n):
        A[k] = A[k] - minA

    # tao mang B gom cac phan tu la 0 va co do dai = max(A)
    B: List[int] = [0] * (max(A) + 1)
    m: int = len(B)

    for i in range(0, n):
        B[A[i]] = B[A[i]] + 1

    for j in range(0, m):
        if (B[j] > 0):
            count = count + 1

    return count
```

- Do trong quá trình chạy tạo thêm mảng phụ B nên có độ phức tạp không gian là $O(n)$.
- Nếu dựa theo thuật toán trên mảng B có tối đa là 10^4 phần tử và mỗi phần tử sẽ nằm trong khoảng $[0, 2.10^4]$. Và thuật toán này chỉ áp dụng cho số nguyên.

Bài 4 (1):

- Đầu tiên ta xem xét giá trị của n, nếu $n < 8$ thì ta sẽ trả về gía trị n, do khi $n < 8$ thì cách phân hoạch duy nhất là phân hoạch thành n số 1.

- Nếu $n > 8$ thì ta tìm số lớn nhất mà n có thể phân hoạch được. Ví dụ nếu $n = 84$ thì số lớn nhất mà n có thể phân hoạch được là 4, do $4^3 = 64 < 84$ nhưng $5^3 = 125 > 84$.

- Cách tìm số lớn nhất này ta chỉ cần lặp i từ $1 \to 100$ (do giới hạn là 10^6 mà 100^3 cũng bằng 10^6). Tạo một biến **largest** tăng thêm 1 sau mỗi lần lặp, nếu lặp đến một số mũ 3 lớn hơn n thì ta thoát lặp, nếu lặp đến số mũ 3 bằng n thì ta được kết quả là 1 (do đó là cách phân hoạch n ít nhất).

- Sau đó ta tạo một mảng phụ để lưu số phần tử của các cách phân hoạch n có thể.

- Sau đó ta giảm dần largest để tìm số phần tử của các lần phân hoạch n, hàm này nhận tham số là n và **largest**. Cách hoạt động như sau:

  - Xét ví dụ là $n = 84$ ta có được **largest** $= 4$.
  - Ta tạo biến **numOfPartition** để lưu giữ số phần tử của lần phân hoạch, ban đầu là 0.
  - Đầu tiên ta thấy $n >$ **largest**^{3} nên ta được phần tử đầu tiên của phần hoạch là 4^3 và n còn lại $84 - 4^3 = 84 - 64 = 20$. Cuối cùng ta cộng **numOfPartition** lên 1.
  - Lần lặp tiếp theo ta thấy $n <$ **largest**^{3} nên ta giảm **largest** xuống 1 còn 3. Tiếp theo ta cũng thấy $n <$ **largest**^{3} nên giảm **largest** xuống 1 còn 2.
  - Lần lặp này ta thấy $n >$ **largest**^{3} nên phần tử tiếp theo của phân hoạch là 2^3 và n còn lại $20 - 2^3 = 20 - 8 = 12$. Lúc này **numOfPartition** được 2. Tương tự lần lặp tiếp theo ta có n còn lại là $12 - 2^3 = 12 - 8 = 4$. Lúc này **numOfPartition** lên 3.
  - Lần lặp cuối ta thấy $n <$ **largest**^{3} nên giảm **largest** xuống còn 1 và vòng while thực hiện xong.
  - Sau khi thoát khỏi vòng while ta thấy n vẫn còn sót lại và $n < 8$ ($n = 4$) nên số phần tử trong phân hoạch còn lại chính là n luôn. (Tức là $4 = 1^3 + 1^3 + 1^3 + 1^3$).
  - Vậy ta có **numOfPartition** $= 7$ và kết quả cuối cùng là $84 = 4^3 + 2^3 + 2^3 + 1^3 + 1^3 + 1^3 + 1^3$.

- Thực hiện xong hàm thì ta thêm số phần tử của phân hoạch với **largest** $= 4$ là 7 vào mảng phụ để lưu.

- Ta giảm **largest** xuống 1 và thực hiện với **largest** $= 3$, tương tự vậy cho đến khi **largest** $= 1$.

- Cuối cùng ta chỉ cần xuất phần tử nhỏ nhất trong mảng phụ đã tạo là xong.

```python
from typing import List

def FindLargest(n: int) -> int:
    """_summary_: Tim so lon nhat ma n co the phan hoach duoc,
   vi du 84 thi se tra ve 4 do 4^3 = 64 < 84 nhung 5^3 = 125 >
   84

```

```
    Args:
        n (int)

    Returns:
        int: so lon nhat
    """
    largest: int = 0

    for i in range(1, 101):
        if (n < i**3):
            break
        if (n == i**3):
            return 1
        largest = largest + 1

    return largest

def FindNumPartition(n: int, largest: int) -> int:
    """_summary_: Tim so phan tu trong phan hoach dua vao largest, vi du largest la 4 thi so phan hoac la 7 do 84 = 4^3 + 2^3 + 2^3 + 1^3 + 1^3 + 1^3 + 1^3

    Args:
        n (int)
        largest (int)

    Returns:
        int: so phan tu trong phan hoach
    """
    numOfPartition: int = 0

    while (largest > 1):
        if (n > largest**3):
            n = n - largest**3
            numOfPartition = numOfPartition + 1
        else:
            largest = largest - 1

    if (n > 0):
        return numOfPartition + n

def FindLeastPartition(n: int) -> int:
    """_summary_: Tim cach phan hoach co luong phan tu la it nhat

    Args:
        n (int)

    Returns:
        int: so luong phan tu it nhat ma co the phan hoach duoc
    """
    if (n < 8):
        return n

    largest: int = FindLargest(n)

    if (largest == 1):
        return largest
```

```python

    minList: List[int] = []

    while (largest > 0):
        minList.append(FindNumPartition(n, largest))
        largest = largest - 1

    return min(minList)

if __name__ == "__main__":
    assert FindLeastPartition(84) == 6 # True
```

- Thuật toán này có thời gian chạy là $O(n^2)$ và do tạo ra một mảng phụ trong quá trình chạy nên độ phức tạp không gian là $O(n)$.