**PHẦN MÔ TẢ BÀI TẬP**

**Mã số học viên:** **M3719017**

**Tên: Phạm Thị Nguyệt Huế**

**Mô tả giải thuật và tính độ phức tạp:**

**Bài 1:**

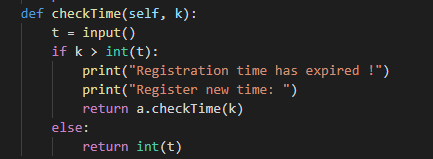
***Mô tả***

Đăng ký hạ cánh của các chuyến bay đảm bảo độ phức tạp là O(logn)

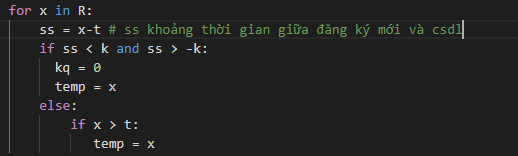
Giả sử có các thời gian đã đăng ký trong csdl lần lược là R = [37,41,46,49,56]

1. Đăng ký thời điểm hạ cánh t, với điều kiện thời điểm này phải cách các thời điểm hạ cánh đã đăng ký ít nhất k phút.:

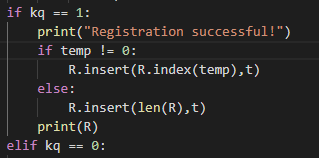
* Kiểm tra thời gian đăng ký: yêu cầu thời gian đăng ký phải sau thời điểm hiện tại (giả sử thời điểm hiện tại là k).



* Kiểm tra thời gian đăng ký phải cách nhau k phút:



+ Nếu thỏa điều kiện cách nhau k phút -> insert thời gian vào csdl (có thứ tự)



1. Trả về thời điểm đăng ký hạ cánh sớm nhất còn lưu trữ trong cấu trúc

* Thời điểm đăng ký hạ cánh sớm nhất còn lưu trữ trong cấu trúc là giá trị đầu tiên trong mảng (dữ liệu được sắp xếp tăng dần):



1. Trả về thời điểm đăng ký hạ cánh trễ nhất còn lưu trữ trong cấu trúc

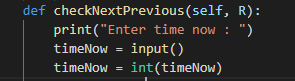
* Thời điểm đăng ký hạ cánh trễ nhất còn lưu trữ trong cấu trúc là giá trị cuối cùng trong mảng (dữ liệu được sắp xếp tăng dần):



1. Trả về thời điểm hạ cánh kế tiếp của thời điểm t cho trước, nếu t là giá trị lớn nhất trong cấu trúc thì -1 được trả về.
2. Trả về thời điểm hạ cánh trước đó của thời điểm t cho trước, nếu t là giá trị nhỏ nhất trong cấu trúc thì -1 được trả về.

* Câu 4, 5 hàm “CheckNextPrevious”

+ Nhập vào một thời điểm t (now)



+ Kiểm tra thời điểm t vừa nhập vào có thuộc R không nếu không thì gọi lại hàm CheckNextPrevious yêu cầu nhập lại.





+ Nếu t thuôc R, get position t

1. Trường hợp nếu position t < length(R)-1
   * + Thời điểm hạ cánh kế tiếp của thời điểm t (timeNext)



* + - Thời điểm hạ cánh trước đó của thời điểm t (timePrevious)



1. Trường hợp nếu position t = length(R)-1
   * + Thời điểm hạ cánh kế tiếp của thời điểm t (timeNext) = -1



1. Trường hợp nếu position t = 0
   * + Thời điểm hạ cánh trước đó của thời điểm t (timePrevious) = -1



1. Có bao nhiêu chuyến bay đã đăng ký hạ cánh từ thời điểm t trở về trước.



***Tính độ phức tạp:***

* (1):
* (2)
* (3)
* (4)
* (5)
* (6)

**Bài 2.**

Giả sử ta có một mảng gồm n phần tử và các phần tử có thể trùng nhau. Viết hàm loại bỏ tất cả các phần tử trùng nhau (giữ lại một giá trị) trong mảng. Tính độ phức tạp của hàm vừa viết.

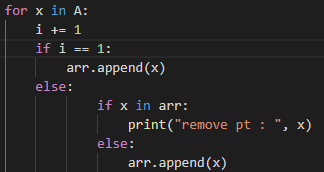
***Mô tả***

Giả sử mảng số là



Khởi tạo một mảng mới arr = [], chạy vòng lặp for A với giá trị là x

* Với giá trị đầu tiên vào mảng arr khởi tạo trước đó.
* Xét điều kiện nếu x đã xuất hiện trong arr thì bỏ qua nó tiếp tục kiểm tra các giá trị tiếp theo, trường hợp nếu x chưa xuất hiện trong mảng mới arr thì thêm vào mảng mới => mảng arr sẽ chứa các phần tử không trùng nhau.



***Tính độ phức tạp:***

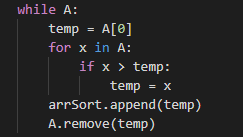
**Bài 3.**

Giả sử ta có một mảng gồm n phần tử và các phần tử không trùng nhau. Viết hàm chọn phần tử lớn thứ k trong mảng.

***Mô tả***

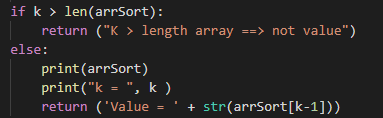
Giả sử mảng nhập vào là A = {101, 11, 3, 4, 50, 69, 7, 8, 9, 0}

* Khởi tạo mảng arrSort = [] sắp xếp lại mảng A theo chiều giảm dần



* Nhập k từ bàn phím (thứ tự lớn thứ k trong mảng), nếu phần từ k lớn hơn độ dài của mảng thì thông báo “not value” ngược lại thì hiển thị phần tử lớn thứ k trong mảng:





***Tính độ phức tạp:***

**Bài 4.**

***Mô tả***

***Tính độ phức tạp:***

**Bài 5.**

Tam giác số.

Cho tam giác số có dạng sau:

A1,1

A2,1 A2,2

A3,1 A3,2 A3,3

....

An,1 An,2 An,3 .... An,n

Tìm một đường đi từ đỉnh A1,1 đến một phần tử nào đó trên cạnh đáy của tam giác sao cho tổng các số trên đường đi là lớn nhất.

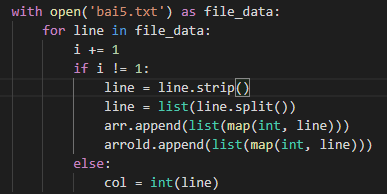
Biết rằng:

* Từ Ai,1 có thể đi đến Ai+1,1 hoặc Ai+1,2
* Từ Ai,j (j1) có thể đi đến Ai+1,j-1, Ai+1,j hoặc Ai+1,j+1.

Tính độ phức tạp của giải thuật.

***Mô tả***

Đọc file dữ liệu: chuyển về array

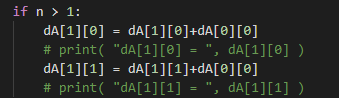


Hàm tính maxSum(dA, n)

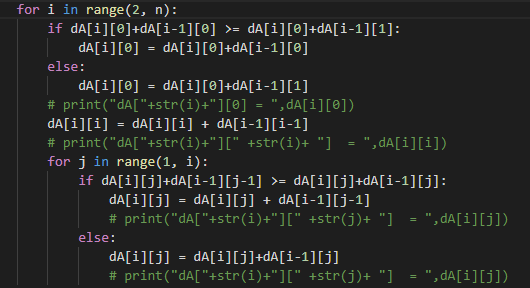
* dA: mảng dữ liệu
* n số hàng dữ liệu tam giác

kiểm tra tam giác > 1 dòng dữ liệu -> gán các giá trị đường đi từ đỉnh đến các phần tử:

* Gán giá trị dòng thứ 2 (2 phần tử):



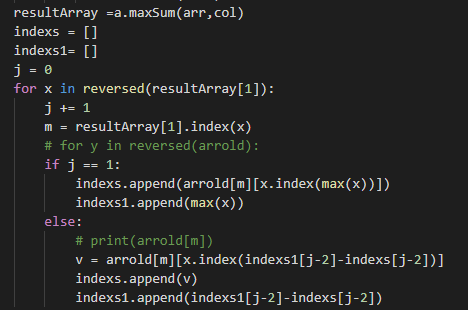
* Chạy vòng lặp for từ dòng thứ 2 -> gán giá trị đường đi trên từng phần tử (với phần tử có nhiều đường đi, gán giá trị có đường đi lớn nhất lên phần tử)



* Trả về kết quả đường đi có giá trị lớn nhất và mảng đã được gán giá trị



Với mảng đã được gán giá trị đường đi trả về -> truy ngược lại tìm giá trị phần tử đường đi (đường đi từ đỉnh đến cạnh đáy có giá trị lớn nhất)



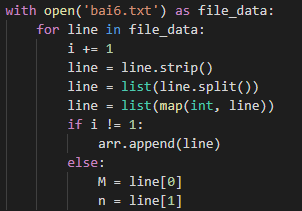
***Tính độ phức tạp:***

**Bài 6.**

Cho một cái ba lô có thể đựng một trọng lượng M và n loại đồ vật, mỗi đồ vật i có một trọng lượng wi và một giá trị pi. Các đồ vật có thể lấy (1) hoặc không lấy (0). Tìm một cách lựa chọn các đồ vật đựng vào ba lô, sao cho tổng trọng lượng không vượt quá M và tổng giá trị là lớn nhất. Tính độ phức tạp của giải thuật.

***Mô tả***

Đọc dữ liệu từ file



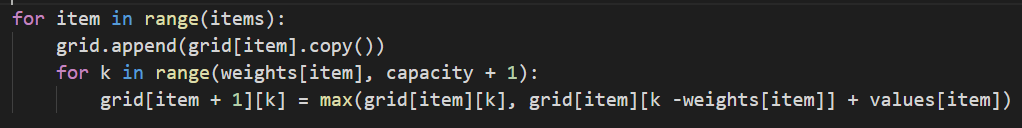
Hàm xử lý knapsack(capacity, weights, values, items)

* Khởi tạo mảng grid có M+1 giá trị



* Chạy vòng lặp for trên số n đồ vật

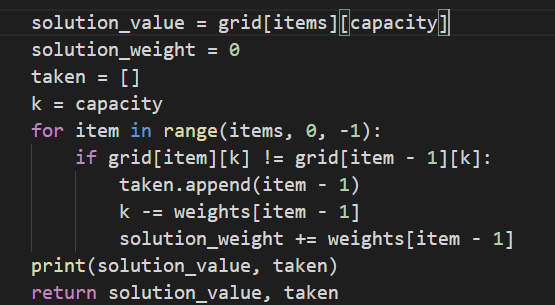
Tính Max(V[i-1,w], V[i-1, wi]+Pi) thêm vào grid



Output grid[]



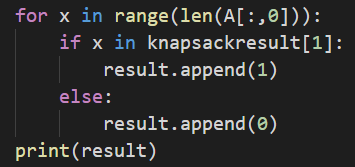
* Lựa chọn đồ vật để tổng khối lượng đạt kết quả cao nhất, trả về tổng khối lượng và đồ vật được chọn



Output: 8 là tổng khối lượng, [3,1] là đồ vật được chọn



Để trả về [0,1,0,1] ( 0 là đồ vật không được chọn, 1 là được chọn)



***Tính độ phức tạp***

**Bài 7.**

Giải bài toán và tính độ phức tạp của giải thuật trong trường hợp bài toán cái ba lô với lựa chọn có thể không nguyên (fractional solution).

***Mô tả***

Giả sử dữ liệu đầu vào như bài 6



Hàm tính fractional\_knapsack(value,weight, capacity)

* Tính tỉ lên (ratio)



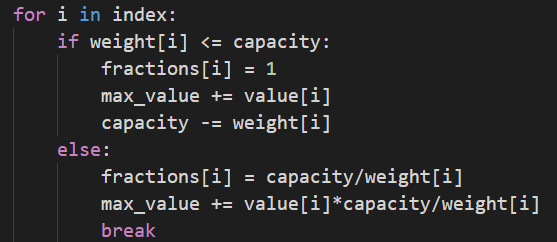
Output



* Tính số đồ vật được chọn ( không nguyên)

+ Nếu trọng lượng của vật thứ i < = trọng lượng Max thì đồ vật dc chọn fractions[i] = 1, giảm trọng lượng Max xuống, tổng giá trị + thêm giá trị đồ vật được chọn.

+ Ngược lại fractions[i] = trọng lượng Max / trọng lượng đồ vật thứ i, tổng giá trị = giá trị đồ vật thứ I \* trọng lượng max/ trọng lượng vật i.



Output: số đồ vật được chọn, tổng giá trị



***Tính độ phức tạp***

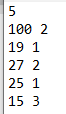
**Bài 8.**

Có n công việc cần phải được thực hiện bởi 1 CPU. Mỗi công việc có 1 deadline di>=1 và lợi tức pi. Mỗi công việc được thực hiện trong 1 đơn vị thời gian. Công việc chỉ đem lại lợi tức khi và chỉ khi chúng được hoàn thành trước deadline tương ứng.

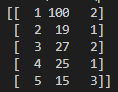
Tìm một tập hợp công việc đem lại tổng lợi tức cao nhất:

***Mô tả***

Xử lý đầu vào:



Trả về



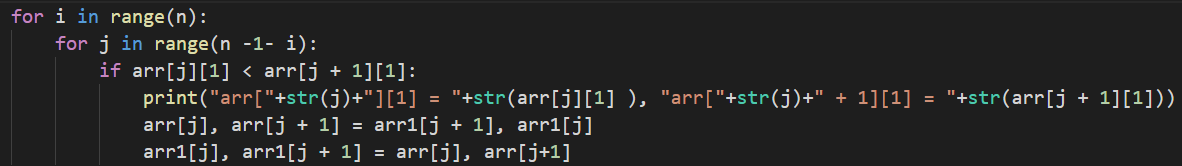
* inputS = [] mảng giá trị bao gồm:

inputS[0] - chỉ số, inputS[1] – lợi tức , inputS[2] – thời gian dealine

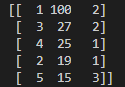
* Thời gian dealine tối đa timedealine = max(inputS[2])
* Số công việc n

Hàm xử lý: JobScheduling(inputS, timedealine, n)

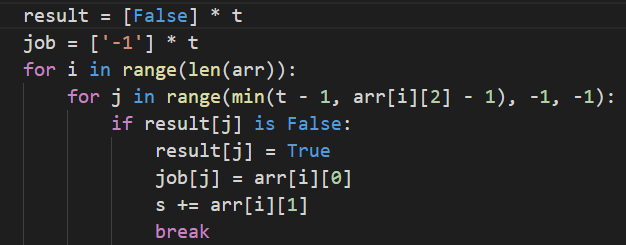
* Sắp xếp mảng theo lợi tức giảm dần



Output:



* Xác định các khoản thời gian dealine, chọn công việc trên từng khoảnj thời gian, nếu cùng 1 thời gian có nhiều công việc chọn công việc có lợi tức cao nhất.



Output: công việc được chọn, tổng lợi tức

