Nguyễn Phi Long - 19521791

Vũ Nguyễn Nhật Thanh – 19522246

BÁO CÁO STRING SEARCH

Nhóm N04

MỘT SỐ THUẬT TOÁN TÌM KIẾM CHUỖI NỔI TIẾNG

Thuật toán vét cạn – ngây thơ

Đây là thuật toán đơn giản nhất. Ý tưởng là ta sẽ quét qua hết tất cả trường hợp. Chuỗi cần tìm có m phần tử, văn bản để tìm kiếm có n phần tử. Ta sẽ duyệt lần lượt các phần tử trong văn bản(n) và các phần tử có trong chuỗi con rồi so sánh (m). Và độ phức tạp là O(m*n)

Thuật toán Rabin – Karp

Thuật toán Rabin – Karp hay thuật toán Karp – Rabin là một thuật toán tìm kiếm chuỗi được tạo ra bởi Richard M. Karp và Michael O. Rabin (1987) sử dụng phép băm để tìm một chuỗi con khớp chính xác với một chuỗi input trong văn bản.

Ý tưởng chính là cũng giống như so sánh kiểu vét cạn. Đầu tiên ta sẽ tính giá trị băm của chuỗi truyền vào. Ta sẽ duyệt hết chuỗi trong văn bản(chuỗi con này sẽ có độ dài bằng chuỗi truyền vào), mỗi lần sẽ dịch chuyển 1 vị trí kí tự. Mỗi lần so sánh thì ta so sánh giá trị băm của 2 chuỗi nhỏ này, nếu chuỗi nhỏ này có giá trị băm như nhau thì tiếp tục so sánh từng phần tử. Nếu không sẽ dịch chuyển đến chuỗi tiếp theo.

```
function RabinKarp(string s[1..n], string pattern[1..m])
    hpattern := hash(pattern[1..m]);
    for i from 1 to n-m+1
        hs := hash(s[i..i+m-1])
        if hs = hpattern
            if s[i..i+m-1] = pattern[1..m]
                return i
    return not found
```

Vậy điều quan trọng là phải implement hàm hash sao cho với các chuỗi khác nhau sẽ trả về các giá trị khác nhau, để không cần phải so sánh chi tiết từng phần tử.

Độ phức tạp trung bình. Nếu hàm hash mà tốt thì ta chỉ cần hash chuỗi truyền vào 1 lần (O(m)) , duyệt qua từng chuỗi con trong văn bản chứ không cần duyệt qua chuỗi truyền vào(O(n)). Nên độ phức tạp là O(n+m).

Độ phức tạp trong trường hợp tệ nhất: là trường hợp tất cả hàm hash tính toán đều trả về cùng giá trị. Lúc này ta cần duyệt qua từng phần tử trong chuỗi con (m lần) và duyệt qua tất cả các chuỗi con có thể trong văn bản (n-m lần). Thế nên độ phức tạp lúc này là O((n-m)*m).

Thuật toán Knuth–Morris–Pratt

Thuật toán được hình thành bởi James H. Morris và được Donald Knuth phát hiện một cách độc lập sau vài tuần từ lý thuyết automata. Morris và Vaughan Pratt cũng đã xuất bản một báo cáo kỹ thuật vào năm 1970. Sau đó cả ba cùng công bố bài báo vào năm 1977 và từ đó nó được gọi là Thuật toán Knuth-Morris-Pratt hay còn gọi là KMP Algorithm.

Ý tưởng của thuật toán là chỉ duyệt qua văn bản (n) và mỗi lần so sánh kí tự của chuỗi con và văn bản mà không trùng thì ta sẽ không lặp lại việc so chuỗi tiếp theo (sau khi dịch phải 1 index) mà sẽ tiếp tục so sánh tại vị trí không trùng.

Ban đầu ta sẽ lần lượt tìm kiếm từng phần tử của string đầu vào với từng phần tử của chuỗi con. Ta lần lượt so sánh từ trái sang phải và dịch chuyển index từ trái sáng phải. Nếu có vị trị không giống nhau thì ta sẽ dịch chuyển string đầu vào qua phải 1 đơn vị. Nếu từng kí tự trong chuỗi giống nhau thì ta ghi dấu lại. Nếu có 1 kí tự không trùng thì ngay lập tức ta xét xem phần tử gần sát cuối cùng có trùng với phần tử trong string đầu vào không. Nếu có thì dịch phần tử bên trái cùng của chuỗi con đến với phần tử trùng trong văn bản và tiếp tục so sánh. Nếu không thì ta di chuyển toàn bộ string đầu vào qua phải và điểm bắt đầu của string đầu vào trùng với vị trị không trùng trước đó. Trong quá trình so sánh nếu toàn bộ phần tử trong string đầu vào đều trùng với 1 chuỗi con nào đó đang xét thì return về vị trí index.

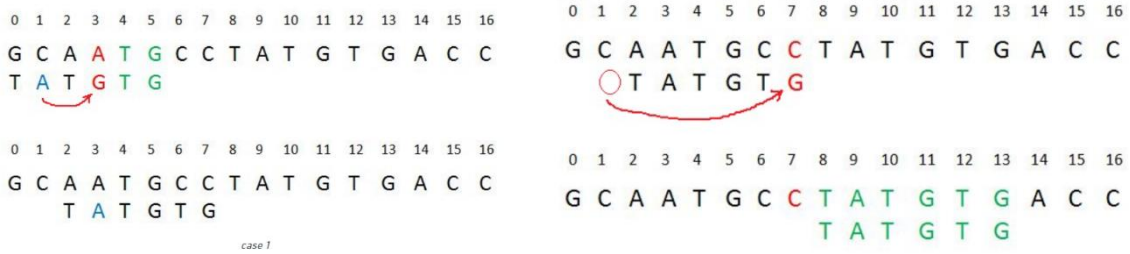
Độ phức tạp này là O(n) cho việc dịch chuyển qua tất cả các phần tử của văn bản.

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an array of integers, P (positions in S at which W is found)
        an integer, nP (number of positions)

    define variables:
        an integer, j ← 0 (the position of the current character in S)
        an integer, k ← 0 (the position of the current character in W)
        an array of integers, T (the table, computed elsewhere)

    let nP ← 0

    while j < length(S) do
        if W[k] = S[j] then
            let j ← j + 1
            let k ← k + 1
            if k = length(W) then
                (occurrence found, if only first occurrence is needed, m ← j - k  may be returned here)
                let P[nP] ← j - k, nP ← nP + 1
                let k ← T[k] (T[length(W)] can't be -1)
        else
            let k ← T[k]
            if k < 0 then
                let j ← j + 1
                let k ← k + 1
```

Thuật toán Boyer – Moore

Thuật toán tìm kiếm chuỗi Boyer – Moore là một thuật toán tìm kiếm chuỗi hiệu quả, là tiêu chuẩn tiêu chuẩn cho tài liệu tìm kiếm chuỗi thực tế và được sử dụng trong nhiều phần mềm nổi tiếng vid dụ grep …..Nó được phát triển bởi Robert S. Boyer và J Strother Moore vào năm 1977.

Ý tưởng chính của bài toán là bỏ qua các lần dịch chuyển nhiều nhất có thể dựa vào quy luật dịch chuyển. String đầu vào và chuỗi con sẽ lần lượt so sánh từ phải sáng trái và dịch chuyển từ trái sang phải. Khác với các thuật toán trước đó đều so sánh và dịch chuyển từ trái sang phải. Kết hợp với bảng tìm kiếm là nguyên nhân làm cho thuật toán này trở nên tối ưu hơn.



case 1

Ta sẽ duyệ từ cuối chuỗi đến đầu chuỗi đầu vào. Nếu phần tử đang xé không trùng khớp nhau thì ta sẽ xét xem phần tử đang xét trong văn bản đó(phần tử bị lệch) có nằm trong chuỗi đầu vào không(sử dung bảng tìm kiếm O(1)). Nếu có tồn tại thì ta dịch chuyển String đầu vào sao cho phần tử đó kí tự mà nằm bên trái cùng sẽ nằm cùng vị trí với phần tử bị lệch. Với cách này thì ta sẽ bỏ qua được một số lần so sánh lãng phí. Và vì ta chọn phần tử bên trái cùng mà khớp với phần tử bị lệch nên chắc chắn sẽ không để lọt trường hợp chuỗi. Còn nếu phần tử bị lệch không nằm trong chuỗi input thì chỉ cần đơn giản dịch chuyển chuỗi vào qua ngay bên phải của phần tử bị lệch.(hình bên phải).

Độ phức tạp thuật toán: Với trường hợp tốt nhất thì tất cả các phần tử là khác nhau, nghĩ là ta chỉ cần dịch chuyển và thực hiện so sánh n/m lần. Vì vậy O(n/m)

Với trường hợp xuất nhất thì các phần tử trong văn bản là giống nhau và các phần tử trong chuỗi đầu vào cũng giống nhau thì mỗi lần phải dịch chuyển 1 lần. Nên độ phức tạp phải là O(n*m).

TÌM HIỂU VỀ PHẦN MỀM GREP

Lịch sử

Grep là một phần mềm tìm kiếm một hoặc nhiều chuỗi con hoặc các chuỗi có đặc trưng cho trước trong 1 file văn bản thuần. Grep ban đầu được phát triển cho hệ điều hành Unix nhưng gần đây thì có các phiên bản trên hệ điều hành gần của Unix. Tác giả của grep là Ken Thompson và được lập trình bởi AT&T Bell Laboratories. Ban đầu thì có 2 phiên bản là fgrep và egrep. 2 phiên bản này dùng để tìm kiếm các string được fix cứng, có nghĩa là tìm kiếm chính xác string đó và tìm kiếm 1 string thõa mãn một syntax cho trước. Và sau này thì cả 2 được gộp lại thành grep. Và mặc định thì grep sẽ xuất ra tất cả các dòng chứa các substring ta muốn.

Thuật toán phía sau phần mềm

Thuật toán: Boyer-Moore algorithm, hạn chế so sánh và tìm kiếm các phần tử dư thừa.Tìm kiếm từ kí tự cuối và sử dụng bảng tìm kiếm.

GNU grep cũng giải phóng vòng lặp bên trong của Boyer-Moore và thiết lập
các mục nhập bảng delta Boyer-Moore theo cách mà nó không cần thực hiện kiểm tra thoát khỏi vòng lặp ở mỗi bước chưa được thực hiện. Kết quả điều này giúp cho GNU grep chạy trung bình ít hơn 3 x86 các instructor thực thi cho mỗi byte đầu vào mà nó thực sự nhìn vào (và nó hoàn toàn bỏ qua nhiều byte).
GNU grep sử dụng lệnh gọi hệ thống đầu vào Unix thô và tránh sao chép dữ liệusau khi đọc nó.

Và Unix hạn chế tách văn bản thành nhiều dòng bởi vì khi tìm kiếm 1 dòng mới nó sẽ phải đi tìm kiếm.

Thao tác sử dụng grep

```
grep [options] pattern [files]
```

Trong đó options là các lựa chọn để thực thi với các yêu cầu khác nhau và files là tên file để tìm kiếm.

Các option ta có thể lựa chọn

-c : hiện ra tổng số hàng chứa yêu cầu, không phải số lần tìm thấy yêu cầu
-h : hiện thị hàng mà yêu cầu được tìm thấy
-i : bỏ qua chữ hoa chữ thường, hoa thường thì cũng như nhau
-l : hiện thị duy nhất tên folder
-n : hiển thị hàng mà tìm thấy yêu cầu và số thứ tự của hàng
-v : hiển thị tất cả các hàng mà không tìm thấy yêu cầu
-f file : lấy pattern từ 1 file
-w : khớp chính xác toàn bộ từ trong yêu cầu
-o : hiển thị tất cả các substring thõa yêu cầu trên các hàng khác nhau

- Các nội dung trên đều được bạn Võ Huy Thành giúp đỡ.

1. Giới thiệu String Search:

- Trong khoa học máy tính, các thuật toán tìm kiếm chuỗi, đôi khi được gọi là thuật toán so khớp chuỗi, là một loại thuật toán chuỗi quan trọng cố gắng tìm một vị trí mà một hoặc một số chuỗi (còn gọi là pattern) được tìm thấy trong một chuỗi hoặc văn bản lớn hơn.

- Một ví dụ cơ bản về tìm kiếm chuỗi là khi mẫu và văn bản được tìm kiếm là mảng các phần tử của bảng chữ cái (tập hữu hạn) Σ. Σ có thể là bảng chữ cái tiếng người, ví dụ: các chữ cái từ A đến Z và các ứng dụng khác có thể sử dụng bảng chữ cái nhị phân (Σ = {0,1}) hoặc bảng chữ cái DNA (Σ = {A, C, G, T}) trong tin sinh học.

- Trong thực tế, phương pháp của thuật toán tìm kiếm chuỗi khả thi có thể bị ảnh hưởng bởi mã hóa chuỗi. Đặc biệt, nếu mã hóa có độ rộng thay đổi được sử dụng, thì việc tìm ký tự thứ N có thể chậm hơn, có lẽ đòi hỏi thời gian tỷ lệ với N. Điều này có thể làm chậm đáng kể một số thuật toán tìm kiếm. Một trong nhiều giải pháp khả thi là thay vào đó tìm kiếm chuỗi các đơn vị mã, nhưng làm như vậy có thể tạo ra kết quả khớp sai trừ khi mã hóa được thiết kế đặc biệt để tránh nó.

2. Cấu trúc thư mục:

- Do nhóm em nhớ sai thứ tự yêu cầu của thầy, nên TH1 biến thành TH2 và ngược lại, nên trong toàn bộ file mã nguồn, tên file và tên thư mục của nhóm em bị đảo ngược lại, nên

file có tên TH1 được dùng để test cho TH2, đồng thời các file này cũng chứa kết quả của TH2, và ngược lại.

- Vì thế trong báo cáo, em vẫn sẽ sử dụng quy ước trên, TH1 sẽ được hiểu là TH2 trong yêu cầu của thầy, và ngược lại.

- m bắt đầu từ 2000, có tổng cộng 50 file, kích thước của chuỗi m tăng lên 2000 mỗi file.

- n trong TH1 bắt đầu từ 505000, k = 2000, có tổng cộng 50 file, kích thước của chuỗi m tăng lên 5000 mỗi file.

- n trong TH2 bắt đầu từ 10000000, có tổng cộng 50 file, kích thước của chuỗi m tăng lên 10000000 mỗi file

- k trong TH1 bắt đầu từ 1, bắt đầu từ 100, có tổng cộng 50 file, kích thước của chuỗi m tăng lên 100 mỗi file

3. Cấu hình máy test:

System:
Kernel: 5.12.3-zen1-1-zen x86_64 bits: 64
Desktop: GNOME 40.1
Distro: Garuda Linux

Machine:
Type: Laptop
System: Dell
product: Inspiron 3580.

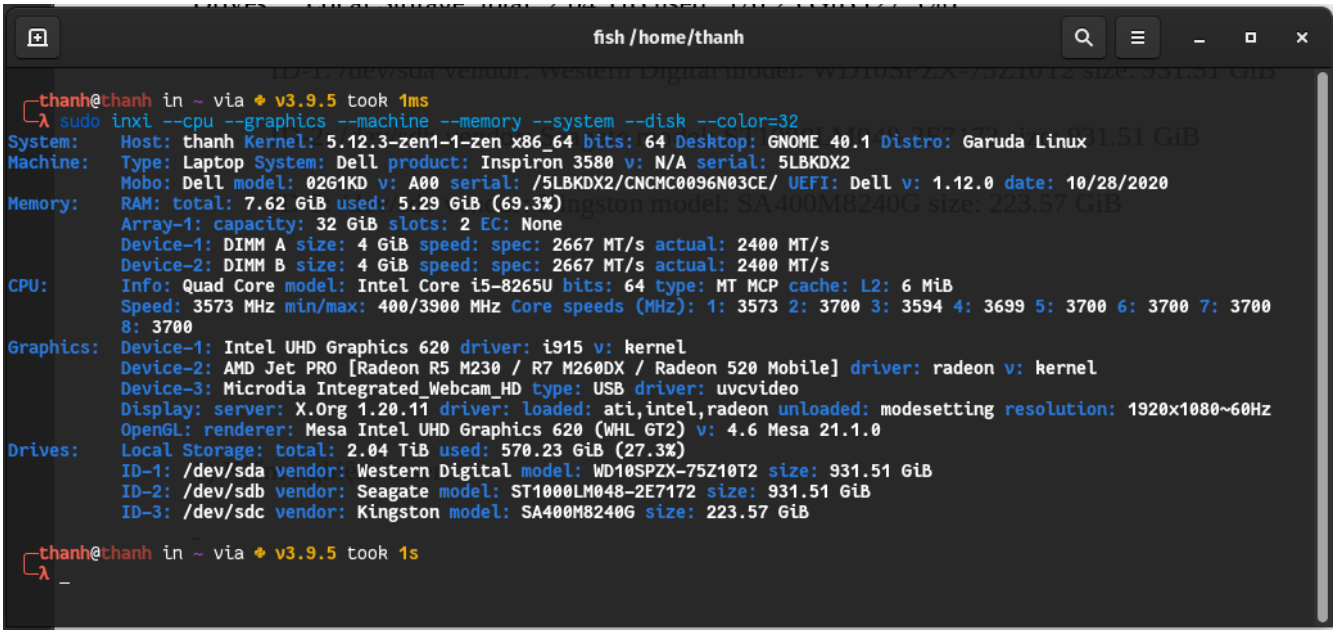
Memory:
RAM: total: 7.62 GiB used: 5.29 GiB (69.3%)
Array-1: capacity: 32 GiB slots: 2 EC: None
Device-1: DIMM A size: 4 GiB speed: spec: 2667 MT/s actual: 2400 MT/s
Device-2: DIMM B size: 4 GiB speed: spec: 2667 MT/s actual: 2400 MT/s

CPU:
Info: Quad Core
model: Intel Core i5-8265U
bits: 64
cache: L2: 6 MiB
Speed: 3573 MHz
min/max: 400/3900 MHz

Graphics:
Device-1: Intel UHD Graphics 620 driver: i915 v: kernel
Device-2: AMD Jet PRO [Radeon R5 M230 / R7 M260DX / Radeon 520 Mobile] driver: radeon v: kernel
Device-3: Microdia Integrated_Webcam_HD type: USB driver: uvcvideo
Display: server: X.Org 1.20.11 driver: loaded: ati,intel,radeon unloaded: modesetting resolution: 1920x1080~60Hz
OpenGL: renderer: Mesa Intel UHD Graphics 620 (WHL GT2) v: 4.6 Mesa 21.1.0

Drives:
Local Storage: total: 2.04 TiB used: 570.23 GiB (27.3%)
ID-1: /dev/sda vendor: Western Digital model: WD10SPZX-75Z10T2 size: 931.51 GiB

ID-2: /dev/sdb vendor: Seagate model: ST1000LM048-2E7172 size: 931.51 GiB
ID-3: /dev/sdc vendor: Kingston model: SA400M8240G size: 223.57 GiB



Dung lượng ổ đĩa lưu trữ: 1TB

- Thương hiệu: Seagate Barracuda
- Tốc độ vòng quay: 5400 RPM
- Bộ nhớ cache: 128MB
- Chuẩn kết nối: SATA III 6Gbps
- Kích thước: 2.5"

4. Thực nghiệm:

- Tạo file test: C++20 để tạo các file chứa chuỗi cần tìm và các file chứa chuỗi lớn, kết hợp với shell cho quá trình thực nghiệm, C++20 được sử dụng để tự động tạo file test và tự động tạo file csv.

- Shell được dùng để hỗ trợ cho việc chạy và đo đạc các thông số như: thời gian chạy, dung lượng ram, nhiệt độ, và dung lượng đĩa cứng, ngoài ra nó được dùng để tự động chỉnh sửa lại các file kết quả mà không lưu được các thông số cần thiết (dung lượng ram) bằng cách chạy lại chỉ một cái testcase lỗi đó.

- Để tránh việc gõ lệnh nhiều lần và giúp tiết kiệm thời gian nhóm em có sử dụng make để tự động hóa việc thử nghiệm.

- Lệnh để tự động hóa quá trình test: **sudo make auto NUM_FILE=50**.

- **NUM_FILE=50** được hiểu là sẽ sử dụng 50 file khác nhau của tham số m cho lần lượt từng file của các tham số n và k, n và k mỗi tham số có số lượng file là 50.

TH2: File chứa ít chuỗi nhưng chuỗi rất dài.

Số lần test = 50*50 = 2500 (lần test)

TH1: File chứa nhiều chuỗi, có độ dài xấp xỉ nhau.

Số lần test = 50*50 + 50*50 = 5000 (lần test)

- Ước tính độ phức tạp: Trong danh sách các thuật toán string search được công bố có tổng cộng 7 độ phức tạp khác nhau (bao gồm cả preprocessing), do thực nghiệm với 3 tham số nên ta sẽ k*n là độ dài của chuỗi được tìm

Danh sách các độ phức tạp:

```
O(n * k * m)
O(m + (k * n + m))
O(m + (k * n - m) * m)
O((m + 62) + n * k / m)
O((m + 62) + n * k * m)
O(m + n * k)
O(m + m * k * n)
```

Do có rất nhiều trường hợp test nên nhóm em sẽ chỉ biểu diễn trên biểu đồ 6 trường hợp ứng với các kích thước m khác nhau

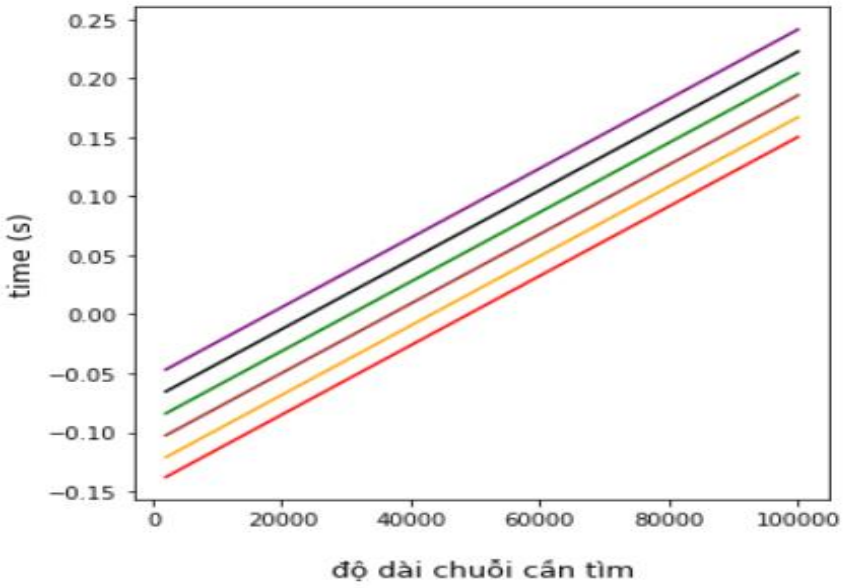
n: 10000000, 100000000, 200000000, 300000000, 400000000, 500000000, mỗi kích thước m sẽ được test với toàn bộ testcase của n (50 file).

k: 100, 1000, 2000, 3000, 4000, 5000, mỗi kích thước m sẽ được test với toàn bộ testcase của k (50 file).

Trong TH2, do kích thước của k khá ít nên nhóm em sẽ không đo độ phức tạp với k mà chỉ đo độ phức tạp với n , toàn bộ 50 file m được test lần lượt với từng file của n (50 file).

TH1: Với n tăng dần nhưng k rất nhiều

Sau quá trình test nhóm em có được kết quả như sau:



n = 505 000 (đỏ)

n = 550 000 (vàng)

n = 600 000 (nâu)

n = 650 000 (xanh)

n = 700 000 (đen)

n = 750 000 (tím)

Nhận xét: Thời gian thực thi tăng dần với n và m cùng tăng (tỉ lệ thuận với m và n).

Nhóm em sử dụng mean square error kết hợp với linear regression để ước lượng độ phức tạp.

```
n * k * m: = 159663.04064680001
m + (k * n + m): = 259.64632472449995
m + (k * n - m) * m: = 18066637.29221933
(m + 62) + n * k / m: = 261.2772935819796
(m + 62) + n * k * m: = 159702.90185646195
m + n * k: = 261.0691879845
m + m * k * n: = 159702.86687532798
```

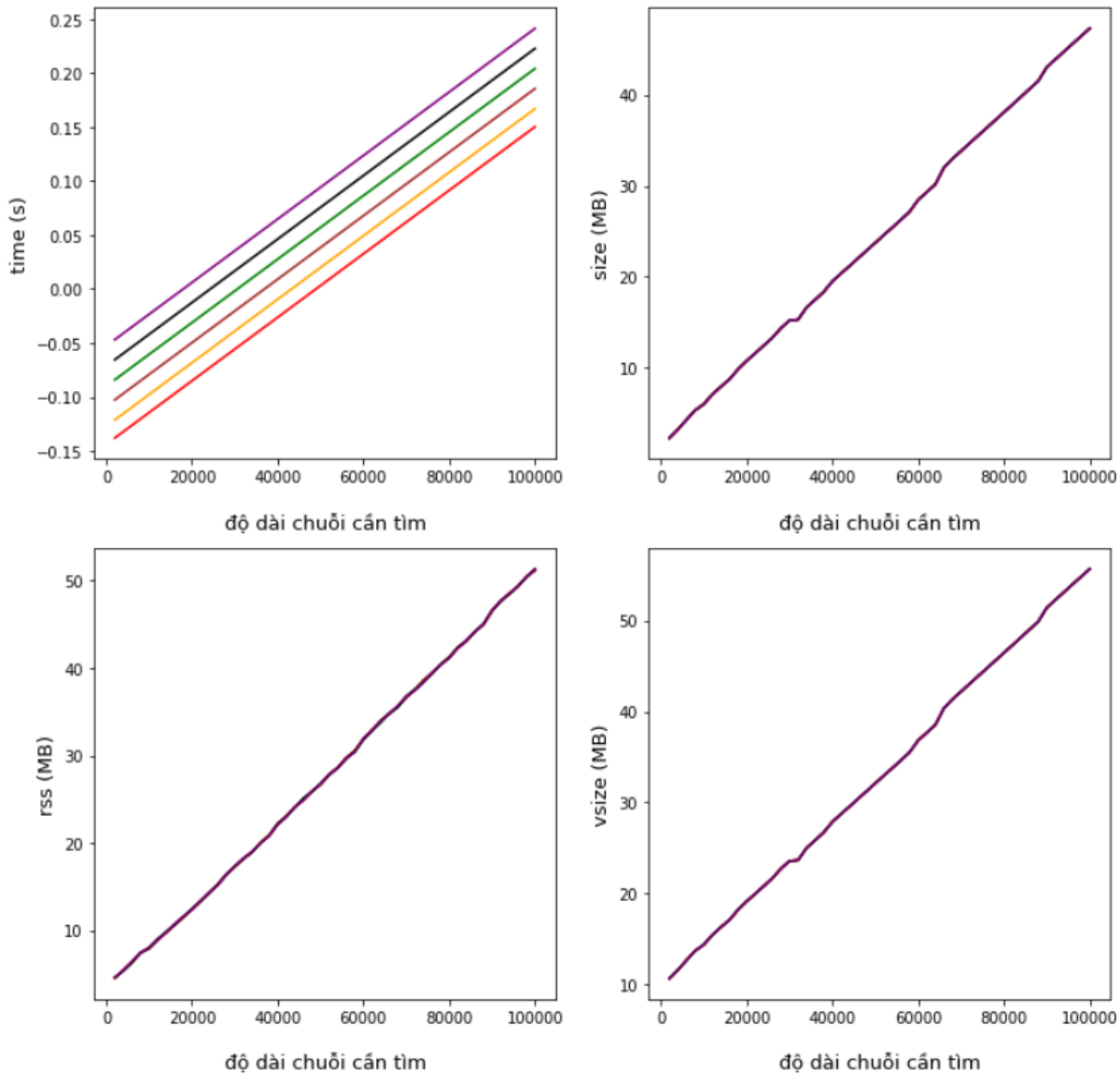
Nhận xét: Với 7 độ phức tạp trên thì độ phức tạp m + (k * n + m) có mean square error là bé nhất, nên độ phức tạp khi ta cố định k và tăng kích thước của n và m là O(m + (k * n + m)).

Với: m là độ dài chuỗi cần tìm.

n là độ dài tối đa của một hàng trong file lớn.

k là số hàng trong file chứa chuỗi lớn.

- Dung lượng ram mà grep sử dụng trong trường hợp này:

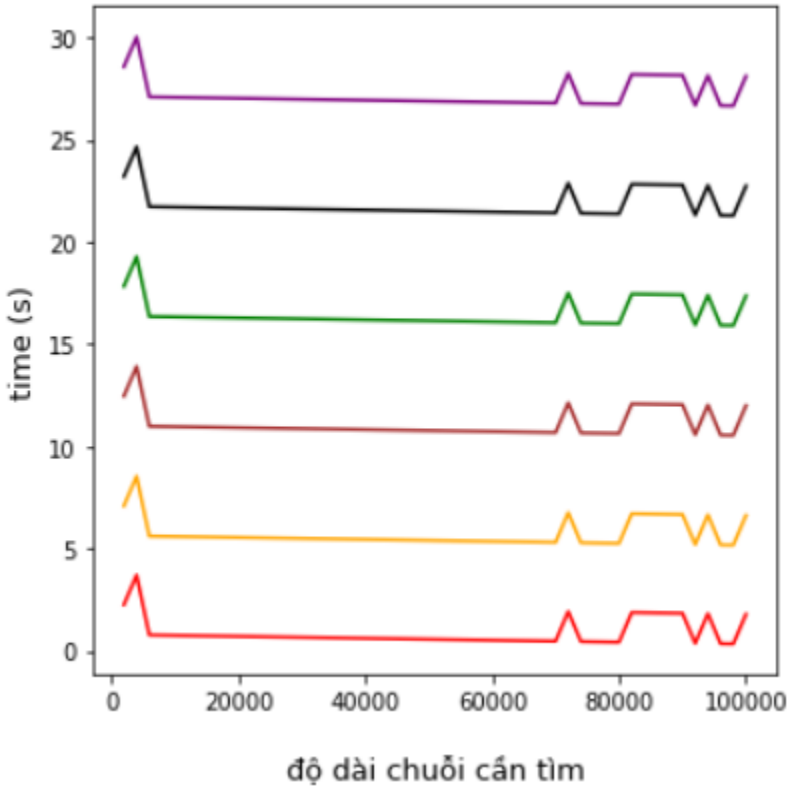


Nhận xét: Dung lượng ram cần cho quá trình tìm chuỗi tăng dần và chỉ phụ thuộc vào m không phụ thuộc vào kích thước của n, biểu hiện là 5 đường thẳng đều trùng nhau.

Kết luận 2: Với n tăng dần và m tăng dần, k cố định (k = 2000) thì thời gian thực thi tăng tỉ lệ thuận với cả m và n, đồng thời dung lượng ram sử dụng chỉ phụ thuộc vào m, trong trường hợp này độ phức tạp của grep là O(m + (k * n + m)).

TH1: Cố định n (400 000), k và m tăng dần.

Sau quá trình test nhóm em có được kết quả như sau:



k = 100 (đỏ)

k = 1000 (vàng)

k = 2000 (nâu)

k = 3000 (xanh)

k = 4000 (đen)

k = 5000 (tím)

Nhận xét: Các đường nằm chồng lên nhau có kích thước lớn dần tính từ dưới lên, chúng đại diện cho các kích thước k khác nhau, suy ra thời gian thực thi tăng dần khi k tăng dần tại một giá trị m, nhưng tại một từng giá trị k ta thấy các đường có xu hướng đi xuống nên suy ra thời gian thực thi giảm dần khi n cố định và m tăng dần (do một đường thẳng tương ứng với 1 k cố định).

Nhóm em sử dụng mean square error kết hợp với linear regression để ước lượng độ phức tạp.

```
n * k * m: = 159663.04064680001
m + (k * n + m): = 259.64632472449995
m + (k * n - m) * m: = 18066637.29221933
(m + 62) + n * k / m: = 261.2772935819796
(m + 62) + n * k * m: = 159702.90185646195
m + n * k: = 261.0691879845
m + m * k * n: = 159702.86687532798
```

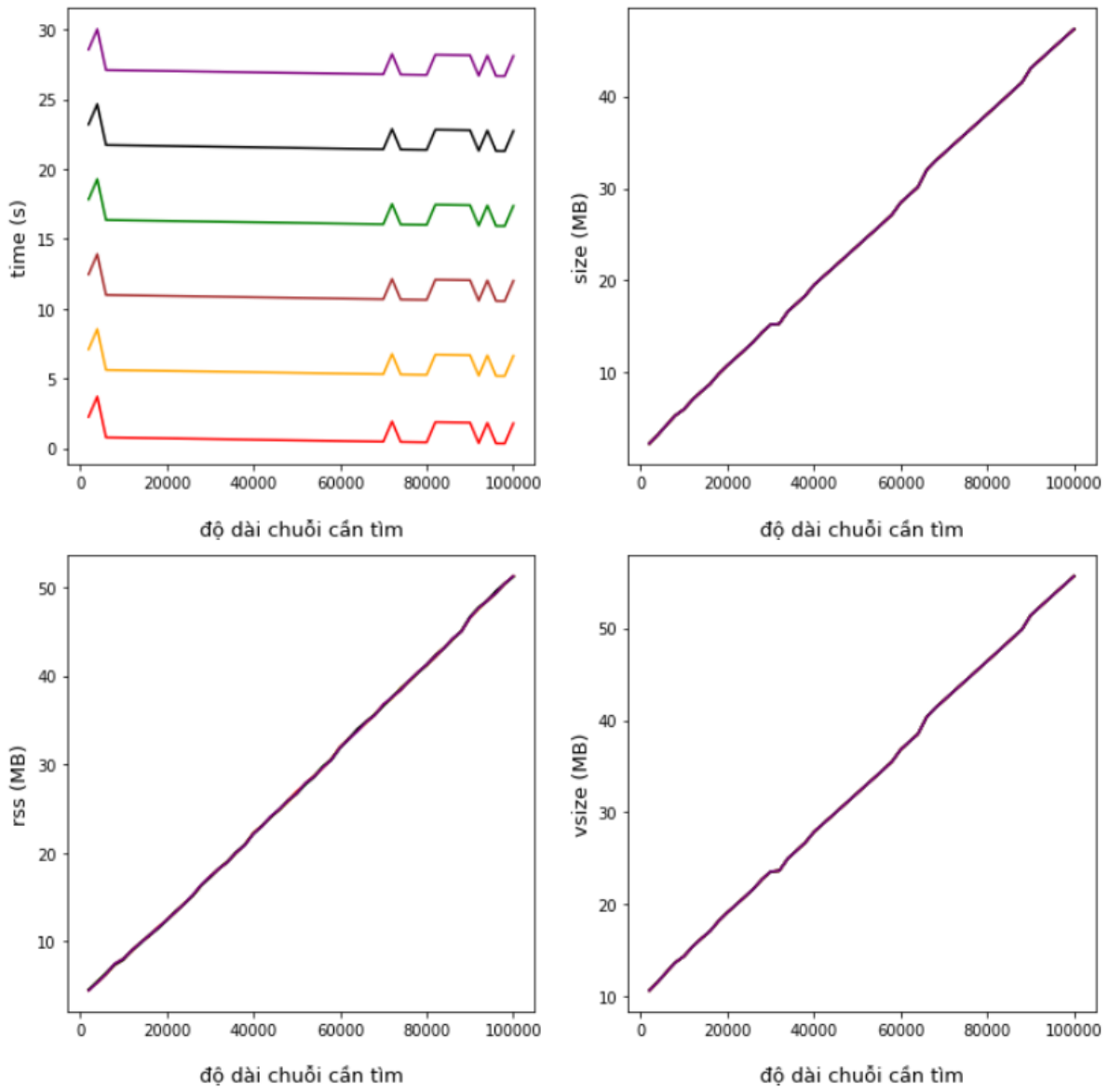
Nhận xét: Với 7 độ phức tạp trên thì độ phức tạp m + (k * n + m) có mean square error là bé nhất, nên độ phức tạp khi ta cố định n và tăng dần kích thước của k và m là O(m + (k * n + m)).

Với: m là độ dài chuỗi cần tìm.

n là độ dài tối đa của một hàng trong file lớn.

k là số hàng trong file chứa chuỗi lớn.

- Dung lượng ram mà grep sử dụng trong trường hợp này:

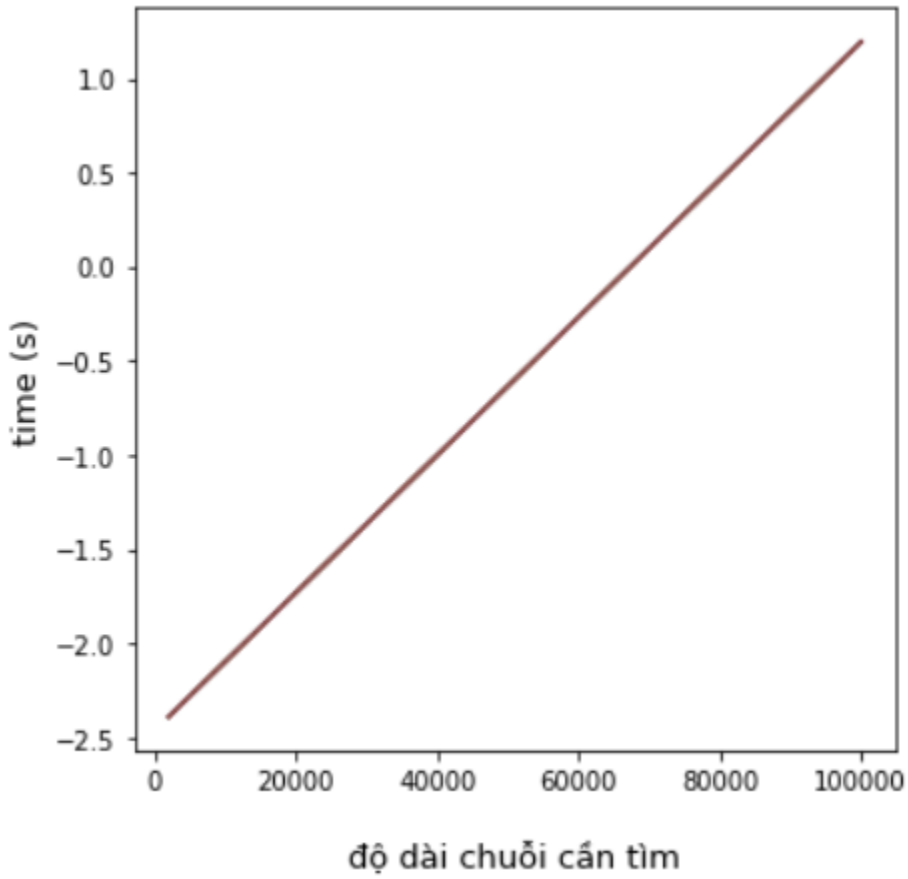


Nhận xét: Dung lượng ram cần cho quá trình tìm chuỗi tăng dần và chỉ phụ thuộc vào m không phụ thuộc vào kích thước của k, biểu hiện là 5 đường thẳng đều trùng nhau.

Kết luận 3: Với k tăng dần và m tăng dần, n cố định (n = 400000) thì thời gian thực thi tăng tỉ lệ thuận với cả m và k, đồng thời dung lượng ram sử dụng chỉ phụ thuộc vào m, trong trường hợp này độ phức tạp của grep là O(m + (k * n + m)).

TH2: cố định k (k = 5), n rất dài và tăng dần

Sau quá trình test nhóm em có được kết quả như sau:



n = 10 000 000 (đỏ)

n = 100 000 000 (vàng)

n = 200 000 000 (nâu)

n = 300 000 000 (đen)

n = 400 000 000 (đen)

n = 500 000 000 (tím)

Nhận xét: Do m quá ngắn khi so với n, với tỉ lệ m/n dao động trong khoảng (1/5000, 1/100), nên thời gian thực thi việc tìm kiếm chuỗi tăng dần.

Nhóm em sử dụng mean square error kết hợp với linear regression để ước lượng độ phức tạp.

```
n * k * m: = 159663.04064680001
m + (k * n + m): = 259.64632472449995
m + (k * n - m) * m: = 18066637.29221933
(m + 62) + n * k / m: = 261.2772935819796
(m + 62) + n * k * m: = 159702.90185646195
m + n * k: = 261.0691879845
m + m * k * n: = 159702.86687532798
```

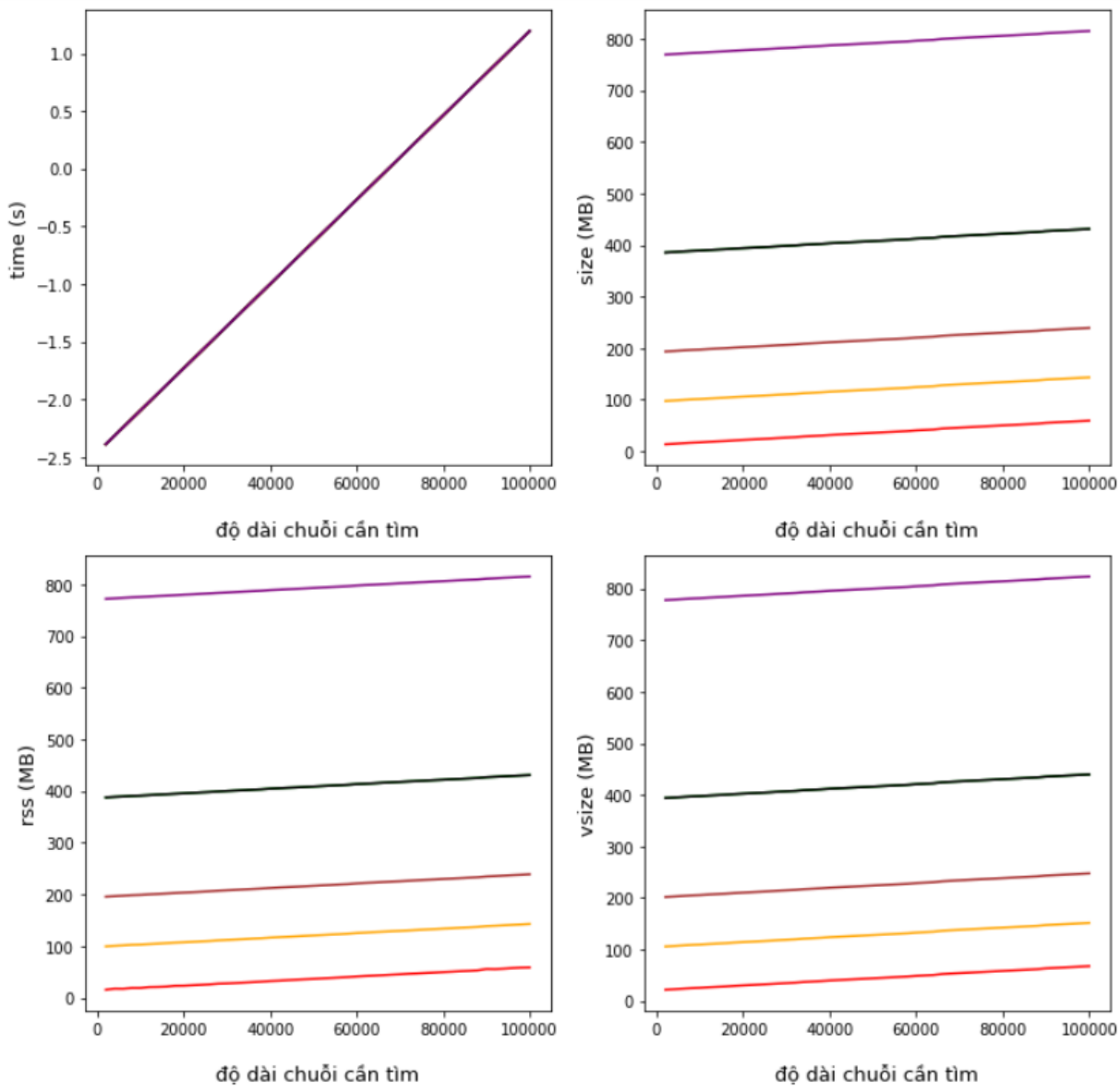
Nhận xét: Với 7 độ phức tạp trên thì độ phức tạp m + (k * n + m) có mean square error là bé nhất, nên độ phức tạp khi mà n lớn hơn rất nhiều so với m là O(m + (k * n + m)).

Với: m là độ dài chuỗi cần tìm.

n là độ dài tối đa của một hàng trong file lớn.

k là số hàng trong file chứa chuỗi lớn.

Dung lượng ram mà grep sử dụng trong trường hợp này:



Nhận xét: Các đường thẳng tăng dần đồng thời vị trí của chúng cũng tăng dần theo kích thước của n, suy ra dung lượng ram tăng dần theo kích thước của m và n.

Kết luận 1: Khi m quá ngắn so với n thì độ phức tạp của grep là O(m + (k * n + m)), khi n tăng thì thời gian thực thi cũng sẽ tăng theo (tỉ lệ thuận với n), và dung lượng ram mà grep cần cũng sẽ tăng (tỉ lệ thuận với m và n).

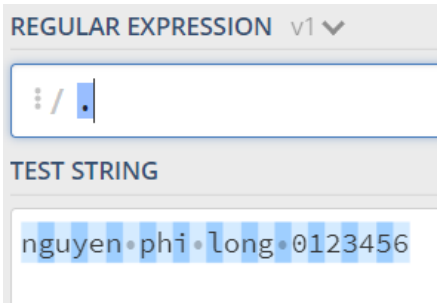
5. Regex:

- Regex là viết tắt của Regular Expression, tên thuần Việt là biểu thức chính quy.

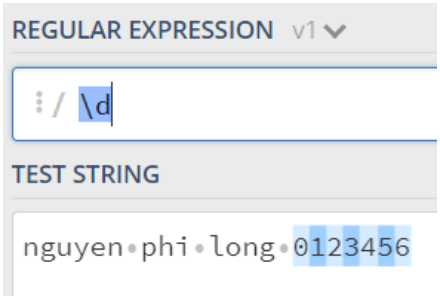
- Là một chuỗi các kí tự miêu tả một bộ các chuỗi kí tự khác, theo những quy tắc và cú pháp nhất định.

- Các cú pháp của Regex:

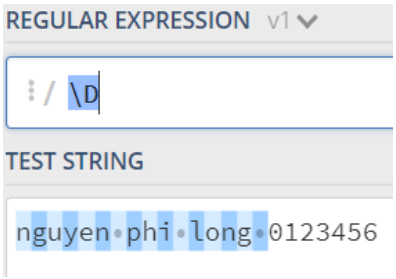
- So khớp cơ bản:
  - **.** : Khớp với bất kỳ kí tự nào.



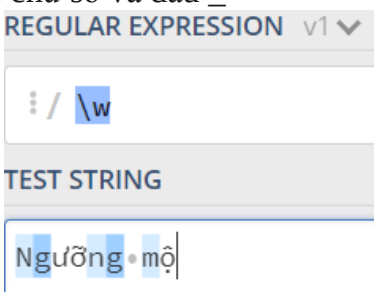
  - **\d** : Khớp với số bất kỳ từ 0-9.



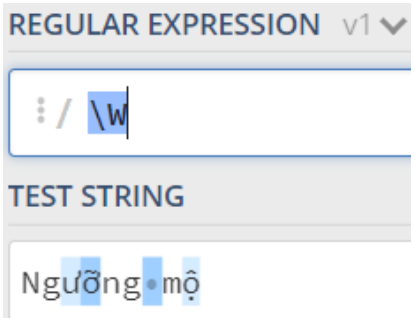
  - **\D**: Phủ định của \d



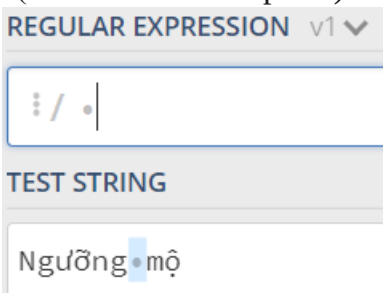
  - **\w**: Khớp với các chữ cái tiếng anh, chữ số và dấu _



  - **\W** : Phủ định của \w.



  - **<dấu cách>**: Khớp với dấu cách(SPACE trên bàn phím).

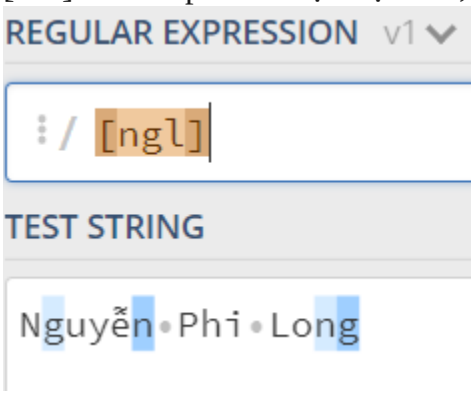


  - **\t**: Khớp với dấu tab.
  - **\n**: Khớp với new line(xuống hàng).

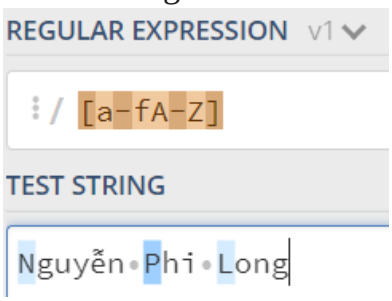
- **\s**: Khớp với dấu trắng bất kì.
- **\S:** Phủ định của \s.

- Kết hợp các chuỗi so khớp:
  - kết hợp các chuỗi so khớp lại với nhau bằng cách đưa chúng vào trong cặp ngoặc vuông.

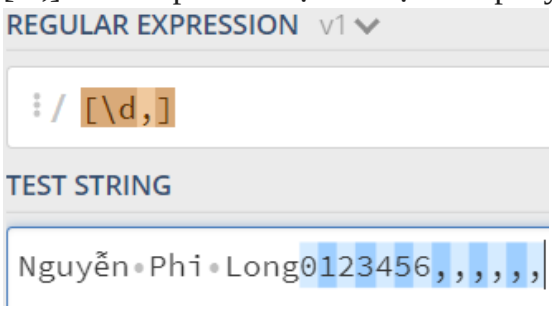
  - [abc] → Khớp các kí tự hoặc là a, hoặc là b, hoặc là c.



  - [a-fA-Z] → a-f là tất cả các kí tự từ a đến f trong bảng chữ cái tiếng anh, A-Z tương tự từ A hoa đến Z hoa. Vậy là regex này khớp với kí tự trong bảng chữ cái tiếng anh bất kể hoa thường.

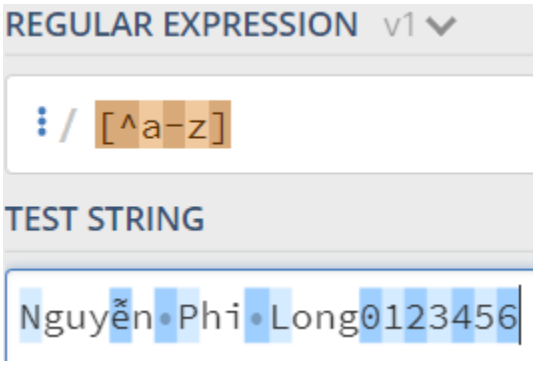


  - [\d,] → Khớp với kí tự số hoặc dấu phẩy.

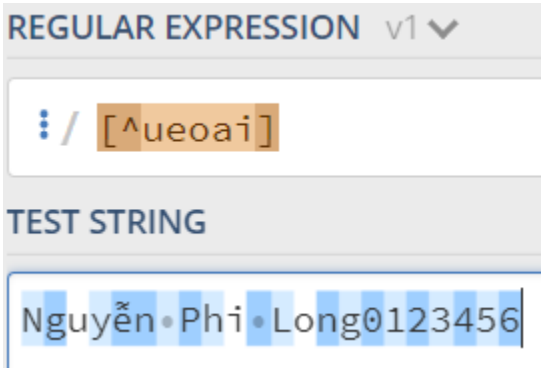


- Có thể loại trừ các giá trị không mong muốn bằng cách:

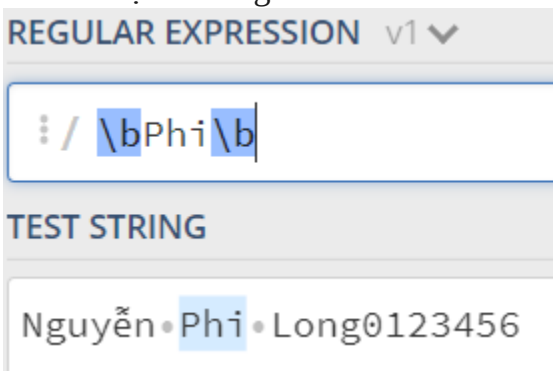
- [^a-z] → Khớp với mọi kí tự trừ các kí tự thường trong bảng chữ cái tiếng anh.



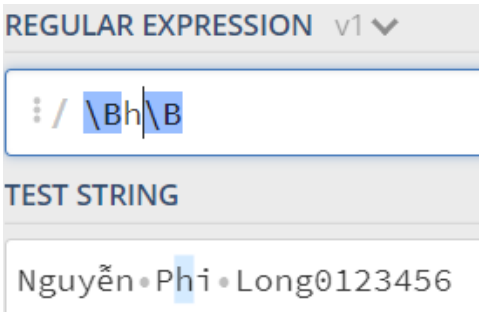
- [^ueoai] → Khớp với mọi kí tự trừ các nguyên âm trong tiếng anh.



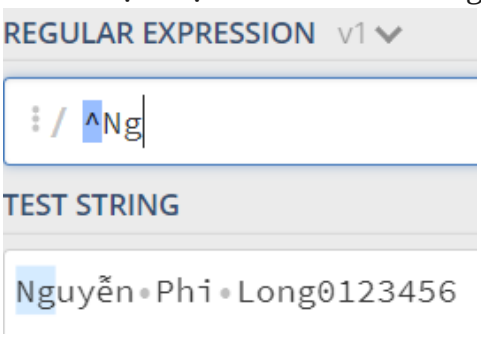
- Các kí tự ranh giới
  - **\b**: Xác định ranh giới của từ.



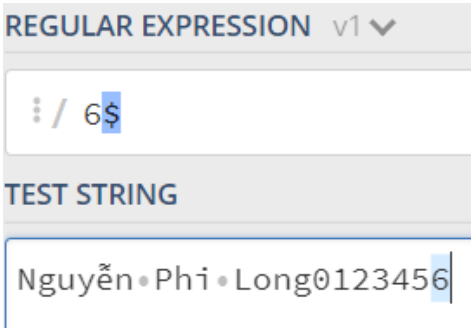
  - **\B:** Bất kì kí tự nào ở giữa của 1 từ.



- ^: Xác định vị trí bắt đầu của dòng.



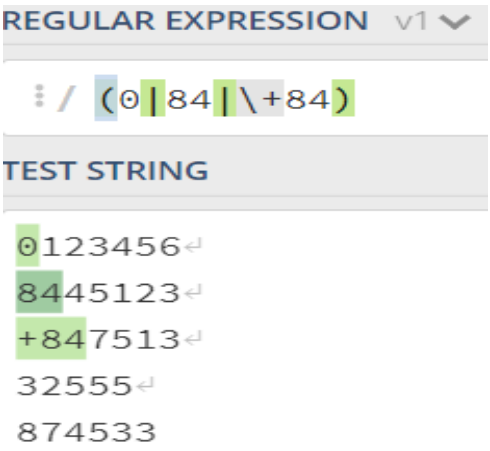
- **$**: Xác định vị trí kết thúc của dòng.



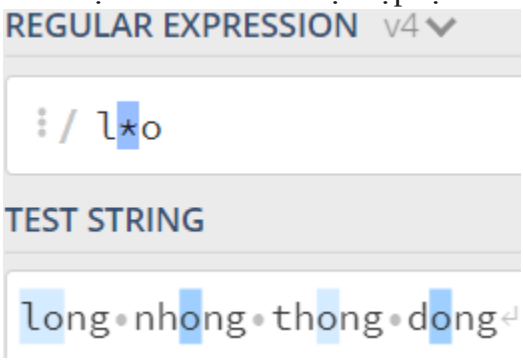
- Sử dụng hoặc trong Regex:

  - Đôi khi ta muốn so khớp hoặc giá trị này, hoặc giá trị kia. Chẳng hạn số điện thoại ở Việt Nam có thể bắt đầu là 0, hoặc 84 hoặc +84.

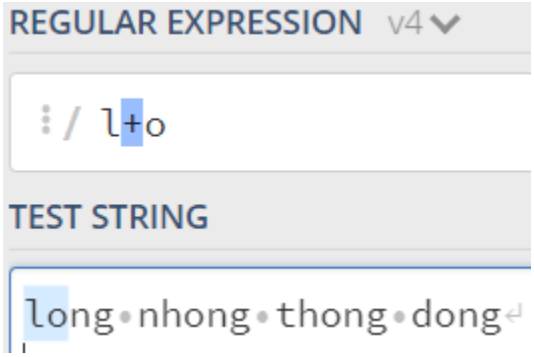
  - Khi đó ta dùng: **(0 | 84 | \+84)** do dấu + là kí tự định lượng nên phải có dấu \ đằng trước.



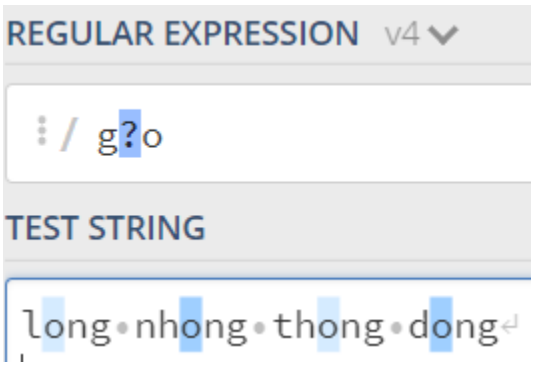
- Các kí tự định lượng:
  - ***** :Kí tự trước có thể được lặp lại 0 hoặc nhiều lần.



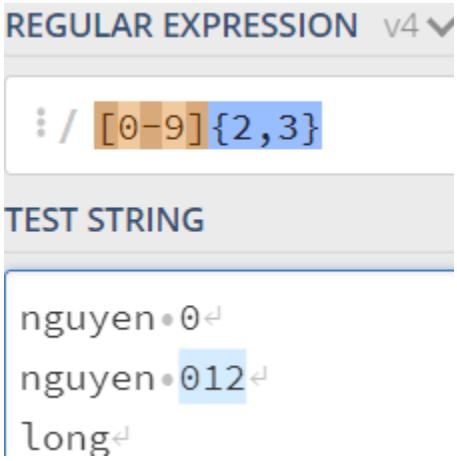
  - **+**: Kí tự trước có thể được lặp lại 1 hoặc nhiều lần.



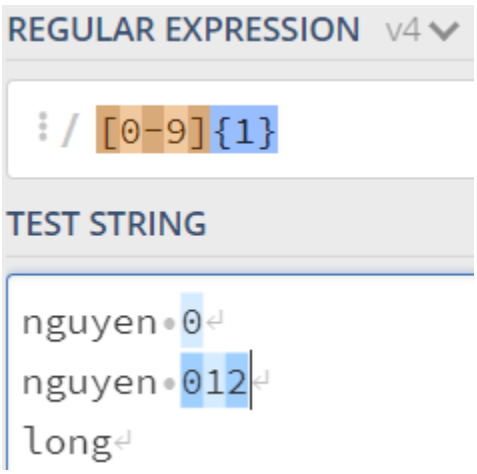
  - **?**: Kí tự trước có thể có hoặc không.



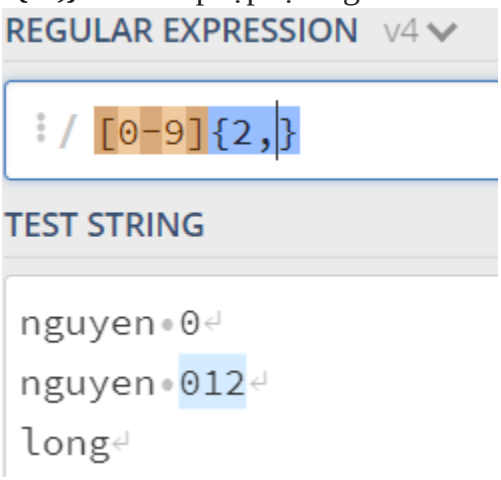
- **{}**: Nó chỉ ra số lần mà một kí tự hoặc một nhóm kí tự lặp lại.
  - **X{m,n}**: So khớp lặp lại regex X với số lần từ m tới n (bao gồm cả m và n).
    ví dụ: [0-9]{2,3} lấy chuỗi có tối thiểu 2 tới 3 kí tự số.



  - **X{m}**: So khớp lặp lại regex X chính xác m lần.

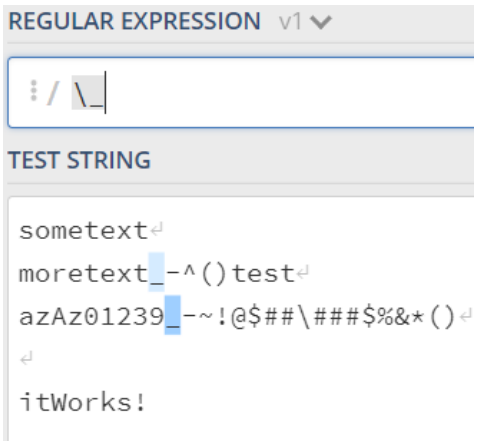


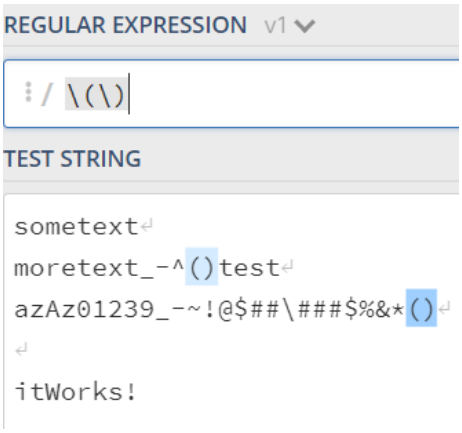
- **X{m,}**: So khớp lặp lại regex X m hoặc nhiều hơn m lần.

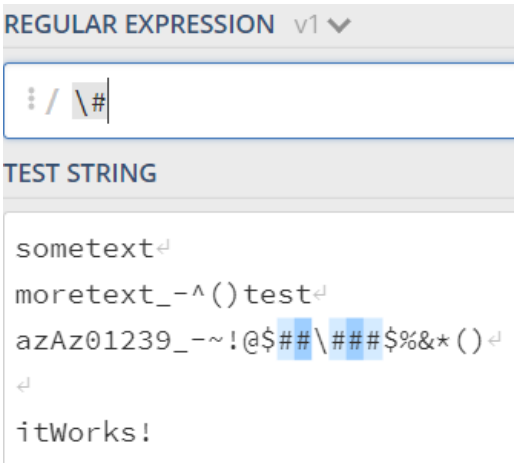


- Các kí tự đặc biệt:

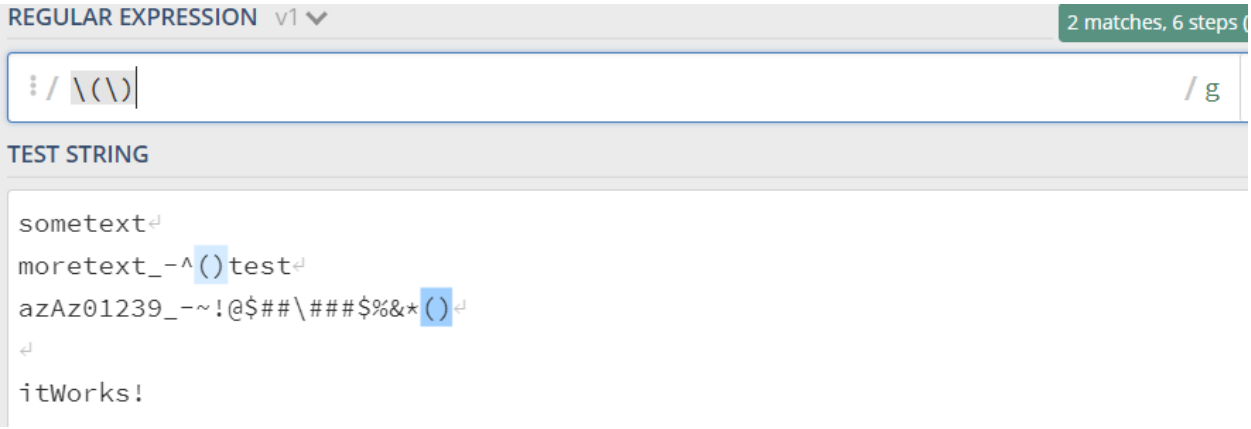
- To match a character having special meaning in regex, you need to use a escape sequence prefix with a backslash (\). E.g., `\.` matches "`.`"; regex `\+` matches "`+`"; and regex `\(` matches "`(`".
- You also need to use regex `\\` to match "`\`" (back-slash).


REGULAR EXPRESSION v1
/ _
TEST STRING
sometext
moretext_-^()test
azAz01239_-~!@$##\###$%&*()
itWorks!


REGULAR EXPRESSION v1
/ \(\)
TEST STRING
sometext
moretext_-^()test
azAz01239_-~!@$##\###$%&*()
itWorks!


REGULAR EXPRESSION v1
/ \#
TEST STRING
sometext
moretext_-^()test
azAz01239_-~!@$##\###$%&*()
itWorks!

Với Regex options là **/g**



Link file Colab:
https://colab.research.google.com/drive/1QTFrCZcbJ9Rga5VXxFS3rtHBewB6GSaA?usp=sharing

Link Github chứa toàn bộ mã nguồn và kết quả: CS523.L21/String_Search at main · Nhat-Thanh/CS523.L21 (github.com)