TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

# NHẬP MÔN PHÂN TÍCH ĐỘ PHỨC TẠP THUẬT TOÁN

## CHỦ ĐỀ: FUZZY SEARCH

**Giảng viên:** Nguyễn Vinh Tiệp

**Lớp**: 19TN

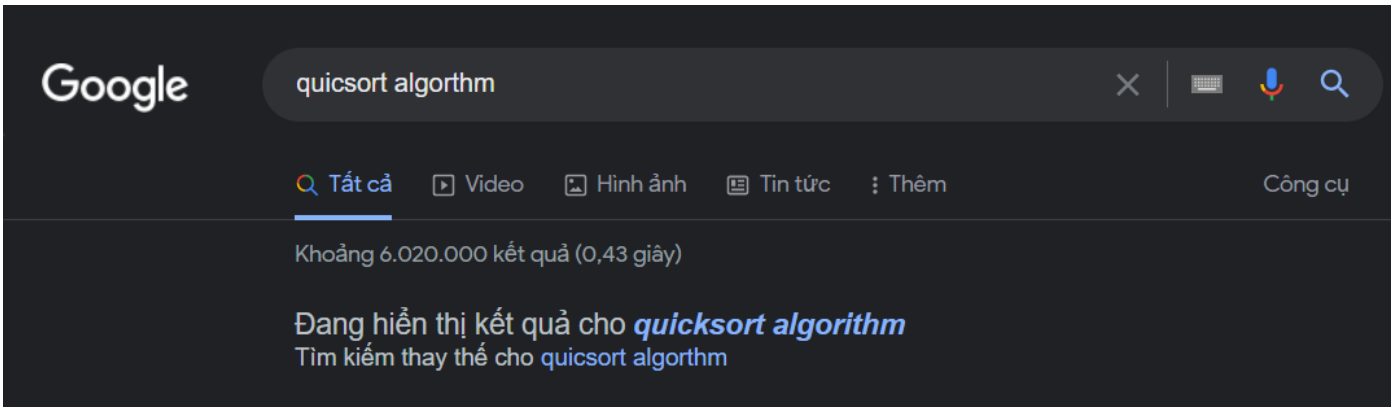
**Người thực hiện:**

| Họ và tên | MSSV |
|---|---|
| Đặng Thái Duy | 19120491 |
| Hà Chí Hào | 19120219 |
| Mai Duy Nam | 19120298 |
| Trần Phương Đình | 19120476 |

# MỤC LỤC

# 1. Định nghĩa bài toán

- Ngữ cảnh đặt ra: Google có lẽ không còn quá xa lạ gì với chúng ta nữa, đây là một trình duyệt phổ biến nhất hiện nay. Vậy chắc ai cũng từng gặp trường hợp gõ một dòng chữ bị sai chính tả, hay không đúng định dạng, … như từ "***quicsort algorthm***" dưới đây. Khi đó Google sẽ có một dòng báo là tìm kiếm thay thế cho chuỗi trên bằng một chuỗi đúng hơn đó là "***quicksort algorithm***" và tìm kiếm dựa trên đó. Vậy Google cũng như các trình duyệt khác đã áp dụng thuật toán gì để có thể làm được điều này. Đó chính là thuật toán Fuzzy Search, ta sẽ đề cập rõ hơn trong bài báo cáo này.




- Định nghĩa: Fuzzy Search (tìm kiếm "mờ"), hay còn hay được gọi là Approximate Search (tìm kiếm "xấp xỉ") là khái niệm để chỉ kỹ thuật để tìm kiếm một xâu "gần giống" (thay vì "giống hệt") so với một xâu cho trước.
- Lợi ích: Việc cài đặt kỹ thuật fuzzy search vào hệ thống sẽ giúp cho người dùng dễ dàng tiếp cận được với nội dung của bài hơn, khi mà họ có thể tìm thấy được những thứ cần thiết, ngay cả khi họ không nhớ được chính xác nội dung mình muốn tìm kiếm là gì.
- Hiện có các thư viện nổi tiếng của Python cho fuzzy search này như: fuzzywuzzy, rapidfuzz, difflib,…

# 2. Hướng tiếp cận

## 2.1. Lý do

Trước hết, ta cần đặt câu hỏi khi nào hai chuỗi bất kỳ được cho là "**gần giống nhau**".

Theo phương pháp thuần túy trước đây, ta chỉ đơn giản so sánh 2 chuỗi bằng cách so khớp tất cả các ký tự bên trong hay không và nhận được kết quả True/False tương ứng.

Nhưng ta cần nhiều hơn thế với Fuzzy Search, vậy vấn đề đặt ra ở đây là phải thay đổi lại khái niệm về việc "gần giống nhau" giữa hai chuỗi a và b, như ta có thể tính ra con số chênh lệnh giữa hai chuỗi và từ đó đề ra 1 con số tối đa đại diện cho sự "gần giống nhau".

Ví dụ thư viện fuzzywuzzy có hàm ratio tính ra độ chênh lệch giữa 2 chuỗi


Cụ thể ta có các ý tưởng sau đây để xem xét:

- Edit distance: Số lượng thao tác để chuyển từ chuỗi a sang chuỗi b
  - Hamming: thay thế ký tự
  - Levenshtein: thay thế, thêm và xóa ký tự
  - Levenshtein-Damerau: thay thế, thêm, xóa và chuyển vị ký tự
  - …
- LCS: chuỗi con chung dài nhất
- Tần suất ký tự
- Đếm số tự thông dụng
- …

→ Tùy thuộc vào ứng dụng hiện tại (so sánh các từ đơn, đoạn văn hay trình tự DNA, …?), một số phương pháp có thể phù hợp hơn các phương pháp khác. Trong nội dung đề cập hiện tại sẽ tập trung giải thích về phương pháp **Edit distance**.

## 2.2. Phương pháp Edit distance

- Hamming distance: số lượng ký tự cần thay thế để chuyển từ chuỗi a sang chuỗi b

  Ví dụ:

  - `d("cat", "hat")` = 1, replace 1st letter (`c -> h`)
  - `d("cat", "lag")` = 2, replace 1st letter (`c -> l`) and 3rd letter (`t -> g`)
  - `d("cat", "cats")` is conventionally not defined (it cannot be achieved by replacing letters)

  Mục đích: thường được dùng sử dụng trong nghiên cứu các mã sửa lỗi - các lược đồ mã hóa có khả năng phục hồi đối với một số bit bị hỏng.

  Tuy nhiên, hạn chế lớn nhất của Hamming là không xử lý tốt các chuỗi bị lệch. Như ví dụ dưới đây cho ra kết quả của việc so sánh 2 chuỗi dưới theo Hamming là 9 nhưng thực tế chỉ là do có sự khác nhau của ký tự khoảng trắng

  ```
  d("Hamming distance",
    "Hammingdistance ") = 9
                 xxxxxxxxx
  ```

- Levenshtein distance:

  Tuy Hamming distance hữu ích trong các ngữ cảnh cụ thể (như mã sửa lỗi), nhưng đối với ngôn ngữ tự nhiên thì nó không phù hợp lắm. Nên ta cần một phương pháp tính Edit distance mạnh mẽ hơn, đó chính là Levenshtein distance.

  Levenshtein distance được sử dụng rộng rãi cũng vì lý do này. Nó có các cách thức sau:

- Thay thế ký tự (như Hamming)
- Thêm ký tự
- Xóa ký tự

Ở đây, khoảng cách được định nghĩa là số lần chỉnh sửa tối thiểu. Có thể có nhiều cách với cùng số lần chỉnh sửa tối thiểu

```
• d("cat", "cats") = 1, as in cats -> cat
• d("moon", "monsoon") = 3, as in monsoon -> mnsoon ->
  msoon -> moon
• d("knight", "knigth") = 2, as in knigth -> knigh ->
  knight
```

Và cùng nhờ vào Levenshtein , ta đã khắc phục được hạn chế của Hamming  nói ở trên trong trường hợp này. Thay vì khoảng cách là 9 ta đã có được kết quả chính xác hơn là 1

```
d("Hamming distance",
  "Hammingdistance") = 1
          insert " ", not replace "d" -> " "!
```

## 3. Trình bày thuật toán

Ta tập trung triển khai phương pháp dùng Levenshtein được đề cập như trên. Vậy làm sao để tính ra được trường hợp với số lần tối thiểu như trên?

Cách đơn giản nhất là thử mọi trường hợp có thể bằng phương pháp đệ quy, nhưng ta có thể tối ưu lên bằng cách dùng phương pháp quy hoạch động với thuật toán Wagner-Fischer. Ngoài ra, còn có một cách tiếp cận khác là dùng “finite automata” nhưng ta sẽ không đề cập trong bài viết này.

### 3.1. Thuật toán Wagner-Fischer

Thuật toán Wagner-Fischer tính toán khoảng cách Levenshtein của hai chuỗi dựa trên việc tính toán một ma trận với giá trị của mỗi ô là khoảng cách Levenshtein của tất cả các tiền tố của chuỗi đầu tiên và các tiền tố của chuỗi thứ hai. Bằng cách lấp đầy ma trận, ta tìm khoảng cách Levenshtein của hai chuỗi đầy đủ là giá trị cuối cùng được tính toán của ma trận.

Ta có mã giả của thuật toán Wagner-Fischer là một hàm tính khoảng cách Levenshtein với input là chuỗi s có độ dài m và chuỗi t có độ dài n, và output là khoảng cách Levenshtein của hai chuỗi như sau:

```
function wagner_fischer(char s[1..m], char t[1..n]):
  // với mọi i và j, d[i][j] sẽ lưu khoảng cách Levenshtein giữa i kí tự đầu tiên của s và j kí tự đầu tiên của t
  // ma trận d có kích thước (m + 1)*(n + 1)
  Declare int d[0..m, 0..n]
  set each element in d to zero

  // chuỗi nguồn có thể chuyển đổi thành chuỗi rỗng bằng cách xóa đi i kí tự
  for i from 1 to m:
      d[i, 0] := i

  // chuỗi rỗng có thể chuyển đổi thành chuỗi đích bằng cách chèn thêm j kí tự tương ứng
  for j from 1 to n:
      d[0, j] := j

  for j from 1 to n:
      for i from 1 to m:
          if s[i] = t[j]:
            substitutionCost := 0
          else:
            substitutionCost := 1

          d[i, j] := minimum(d[i-1, j] + 1,                  // xóa
                             d[i, j-1] + 1,                  // chèn
                             d[i-1, j-1] + substitutionCost)  // thay thế
  return d[m, n]
```

### 3.2. Minh họa thuật toán

*Xét ví dụ: tính khoảng cách Levenshtein của 2 chuỗi s =* ***HORS*** *và t =* ***ROS*** *với m = 4, n = 3*

Ta lập ma trận `d` có kích thước `(m + 1)*(n + 1)`:

| | ‘‘ | H | O | R | S |
|---|---|---|---|---|---|
| ‘‘ | 0 | 0 | 0 | 0 | 0 |
| R | 0 | 0 | 0 | 0 | 0 |
| O | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 |

Điền các kết quả tương ứng ở hàng `0` và cột `0`:

| | ‘‘ | H | O | R | S |
|---|---|---|---|---|---|
| ‘‘ | 0 | 1 | 2 | 3 | 4 |
| R | 1 | 0 | 0 | 0 | 0 |
| O | 2 | 0 | 0 | 0 | 0 |
| S | 3 | 0 | 0 | 0 | 0 |

Ta tính các giá trị của các ô còn lại dựa trên công thức quy hoạch động:

```
d[i][j] :=

min(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] + (s[i]=t[j] ? 0 : 1))
```

| | ‘‘ | H | O | R | S |
|---|---|---|---|---|---|
| ‘‘ | 0 | 1 | 2 | 3 | 4 |
| R | 1 | 1 | 2 | 2 | 3 |
| O | 2 | 2 | 1 | 2 | 3 |
| S | 3 | 3 | 2 | 2 | 2 |

Ta có `d[m][n] = 2` chính là khoảng cách Levenshtein của hai chuỗi a và b.

Mặc khác, ta có thể tiến hành truy hồi để tìm ra các hành động chỉnh sửa từ chuỗi a sang chuỗi b, cụ thể như sau: ta sẽ bắt đầu từ ô `d[m][n]` và đi theo hướng đến ô `d[1][1]`. Ta sẽ chọn ô `d[i][j] := min (d[i-1][ j] + 1, d[i][j-1] + 1, d[i-1][j-1] + (s[i] = t[j] ? 0 : 1))`. Khi số lần chỉnh sửa có sự thay đổi, ta sẽ biết được hành động chỉnh sửa ở bước đó là gì dựa vào ô đã chọn (chèn, xóa, thay thế).

| | ‘‘ | H | O | R | S |
|---|---|---|---|---|---|
| ‘‘ | 0 | 1 | 2 | 3 | 4 |
| R | 1 | 1 | 2 | 2 | 3 |
| O | 2 | 2 | 1 | 2 | 3 |
| S | 3 | 3 | 2 | 2 | 2 |

Lộ trình theo mũi tên màu xanh:

- ROS và HORS: Lúc này ta đi đến ô `d[3][4] = 2` là ô có giá trị nhỏ nhất, tương ứng với hai chuỗi con là RO, HOR (1)
- RO và HOR: ta đi đến ô `d[3][3] = 1` tương ứng với hai chuỗi con RO và HO (2)
- RO và HO: ta đi đến ô `d[2][2] = 1` tương ứng với hai chuỗi con R và H (3)
- R và H: đi đến ô `d[1][1] = 0`. Kết thúc. (4)

Ở đây, ta để ý có hai lần khoảng cách Levenshtein tăng lên, đó là lúc (3) đi đến (4) (hành động thay thế R thành H) và lúc (1) đi đến (2) (hành động chèn R vào cuối HO).

Như vậy, để chuyển đổi chuỗi ROS thành HORS, cần hai bước chỉnh sửa chuỗi ROS như sau:

- Thay thế R thành H: ROS → HOS
- Chèn R vào bên phải HO: HOS → HORS

*Chứng minh tính đúng đắn của thuật toán:*

- Ở hàng 0 và cột 0, thuật toán đúng vì ta có thể chuyển đổi chuỗi `s[1..i]` thành chuỗi `t[1..0] = ''` bằng cách xóa đi i kí tự, tương tự ta cũng có thể chuyển đổi chuỗi `s[1..0]` thành chuỗi `t[1..j]` bằng cách chèn thêm j kí tự.
- Nếu `s[i] = t[j]`, ta có thể chuyển đổi `s[1..i - 1]` thành `t[1..j - 1]` với `k` phép chuyển đổi. (`s[i] = t[j]` nên không cần thao tác nào với hai kí tự này)
- Mặc khác, có ba hướng tiếp cận để chuyển đổi `s[1..i]` thành `t[1..j]`:
  + **CHÈN**: Nếu ta chuyển đổi chuỗi `s[1..i]` thành `t[1..j - 1]` mất `k` phép chuyển đổi, thì ta có thể thêm `t[j]` để được `t[1..j]` (tổng cộng `k + 1` phép chuyển đổi)
  + **XÓA**: Nếu ta chuyển đổi chuỗi `s[1..i - 1]` thành `t[1..j]` mất `k` phép chuyển đổi, thì ta có thể xóa `s[i]` để được `t[1..j]` (tổng cộng `k + 1` phép chuyển đổi)

+ **THAY THẾ**: Nếu ta chuyển đổi chuỗi `s[1..i - 1]` thành `t[1..j - 1]` mất k phép chuyển đổi, thì ta có thể thay thế `s[i]` thành `t[j]` để được `t[1..j]` (tổng cộng k + 1 phép chuyển đổi)

- Số thao tác để chuyển đổi `s[1..n]` thành `t[1..m]` cũng chính là số thao tác để chuyển đổi s thành t. Do đó `d[m][n]` là khoảng cách Levenshtein của hai chuỗi s và t.

## 4. Độ phức tạp của thuật toán

### 4.1. Lý thuyết

Một lần tính khoảng cách Levenshtein bằng Wagner-Fischer:

- Không gian: $O(nm)$
- Thời gian: $O(nm)$ (với n, m là độ dài của 2 chuỗi mình đang tính khoảng cách Levenshtein)

Chứng minh:

Ghi chú: Mục đích của hàm `np.zeroes((m+1, n+1), dtype=int)` là để gán một ma trận có kích thước $(m+1, n+1)$ toàn là số 0. Cho nên ta có thế nói xấp xỉ là hàm đấy có $(n+1) * (m+1)$ phép gán, và 0 phép so sánh.

```
def levenshtein_(a: str, b: str) -> int:
    m, n = len(a), len(b)       2 gán
    declare int d[0..m, 0..n]
    set each element in d to zero           (n+1)(m+1) gán

    for i from 0 to m:          m+1 gán     m+2 so sánh
      d[i, 0] = i               m+1 gán
    for i from 0 to n:          n+1 gán     n+2 so sánh
      d[0, i] = i               n+1 gán

    for j from 1 to n:                n gán       n+1 so sánh
```

```
        for i from 1 to m:              m gán       m+1 so sánh
            cost = a[i-1] != b[j-1] ? 1 : 0             m gán  m so sánh
            d[i, j] = min(d[i-1, j-1] + cost,           m gán 2m so sánh
                          d[i, j-1]   + 1,
                          d[i-1, j]   + 1)
    return d[m, n]
```

Tổng số phép gán:

$G = 2 + (n+1)(m+1) + 2 * (m+1) + 2 * (n+1) + n + n * 3m$

$= 2 + nm + n + m + 1 + 2m + 2 + 2n + 2 + n + 3nm$

$= 4nm + 4n + 3m + 7$

Tổng số phép so sánh:

$S = m + 2 + n + 2 + n + 1 + (n+1) * (4m+1) = 4nm + 3n + 5m + 6$

→ **Số phép toán:** $\boldsymbol{S + G = 8nm + 7n + 8m + 13 = O(nm)}$

Với không gian thì ta chỉ tạo thêm 1 mảng 2 chiều có kích thước $n * m$ cho việc tính toán quy hoạch động, nên độ phức tạp không gian cũng là $O(nm)$.

Gọi số chuỗi trong tập dữ liệu mình có là z. Độ phức tạp của Fuzzy Search:

Không gian: $O(znm)$

Thời gian: $O(znm)$

Vì ta chỉ đang thực hiện tính toán khoảng cách Levenshtein giữa chuỗi tìm kiếm và tất cả mọi chuỗi trong tập dữ liệu (có z chuỗi như vậy) cho nên độ phức tạp sẽ là $O(znm)$, nhưng lúc này m

là độ dài trung bình của các chuỗi trong tập dữ liệu. Còn việc sắp xếp lại các khoảng cách có độ phức tạp là $O(zlog(z))$, là không đáng kể so với $O(znm)$.

## 4.2. Thực nghiệm

### a. Cài đặt thuật toán

Dưới đây là cài đặt của thuật toán Wagner-Fischer sử dụng ngôn ngữ Python. Cách cài đặt khá tương đồng so với mã giả ở phía trên.

```python
def levenshtein_(a: str, b: str) -> int:
    m, n = len(a), len(b)
    d = np.zeros((m+1, n+1), dtype=int)

    for i in range(m+1):
      d[i, 0] = i
    for i in range(n+1):
      d[0, i] = i

    for j in range(1, n+1):
        for i in range(1, m+1):
            cost = 1 if a[i-1] != b[j-1] else 0
            d[i, j] = min(d[i-1, j-1] + cost,   # substitute
                          d[i, j-1]   + 1,      # insert
                          d[i-1, j]   + 1)      # delete
    return d[m, n]
```

Dưới đây là cài đặt của hàm tìm chuỗi gần giống nhất với chuỗi query $q$ từ một tập dữ liệu các chuỗi. Ta tìm khoảng cách Levenshtein từ chuỗi query $q$ đến từng chuỗi trong tập dữ liệu, sau đó tính score và chọn chuỗi nào có score cao nhất. Score được tính theo công thức:

$$\text{score}(\text{q}, \text{t}) = 1 - \frac{\text{d}(\text{q}, \text{t})}{\max(\text{q.length}, \text{t.length})}$$

Việc tính score như vậy nhằm mục đích "scale" giá trị khoảng cách Levenshtein về khoảng 0 đến 1. Vì giữa hai chuỗi $a$ và $b$ bất kỳ, khoảng cách Levenshtein tối thiểu là $d(a, b) = 0$ (khi đó chuỗi $a$ và $b$ giống nhau hoàn toàn), và tối đa là $d(a, b) = \max(a.\text{length}, b.\text{length})$, cho nên việc tính score theo công thức này cho biết hai chuỗi $a$ và $b$ giống nhau đến bao nhiêu phần

trăm. So sánh bằng tỷ lệ phần trăm như vậy là hợp lý hơn so với so sánh trực tiếp bằng khoảng cách Levenshtein.

```
def levenshtein_find(query_text, texts):
    best_score = -np.inf
    best_result = ''

    for text in texts:
        if len(text) == 0:
            continue

        text = text.lower()
        d = levenshtein_(query_text, text)
        score = 1.0 - d / max(len(query_text), len(text))

        if score > best_score:
            best_score = score
            best_result = text

    return (best_score, best_result)
```

### b. Chạy thử thuật toán với một số ví dụ thực tế

Phần thực nghiệm đo thời gian chạy của thuật toán sử dụng tập dữ liệu các thành phố trên thế giới. Tập dữ liệu có kích thước là 39.186 thành phố, với tên các thành phố đều được đưa về dạng lowercase.

```
df = pd.read_csv("data.csv")
cities = df["city_ascii"].apply(lambda x: x.lower()).unique()
print(cities)
print(cities.size)
```

```
['tokyo' 'jakarta' 'delhi' ... 'nord' 'timmiarmiut' 'nordvik']
39186
```

Bảng dưới đây so sánh kết quả và thời gian chạy thuật toán với một số query khác nhau. Với mỗi query thuật toán được chạy ba lần. Thời gian chạy của thuật toán được tính bằng giây.

| Query | Lần chạy 1 | Lần chạy 2 | Lần chạy 3 | Kết quả trả về |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| ho chiminj | 8.509063 | 8.484985 | 5.811747 | ho chi minh city |
| hanpi | 3.024003 | 3.027255 | 2.982788 | hanoi |
| danang | 3.525662 | 3.564528 | 3.593004 | danyang |
| daklk | 2.992620 | 2.967505 | 2.973129 | dakar |

Có thể thấy, chuỗi query càng dài thì thuật toán càng mất nhiều thời gian chạy. Trong số bốn query ở trên thì có hai query là "ho chiminj" và "hanpi" trả về kết quả như ta mong đợi là "ho chi minh city" và "ha noi". Hai query còn lại cho ra kết quả khác với mong muốn ("danyang" thay vì "da nang" và "dakar" thay vì "daklak"). Tuy nhiên điều này là do ta chỉ đang thu thập một chuỗi kết quả tốt nhất được trả về thôi, trong khi trong thực tế có thể có nhiễu chuỗi cho ra score bằng nhau. Bảng dưới đây thể hiện điểm của 8 kết quả tốt nhất khi ta tìm với query "danang".

| **Điểm** | **Kết quả** |
|---|---|
| 0.857143 | danyang |
| 0.857143 | da nang |
| 0.833333 | datang |
| 0.750000 | dangyang |
| 0.714286 | dandong |
| 0.714286 | kananga |
| 0.714286 | daxiang |
| 0.714286 | kantang |

## c. Kiểm tra độ phức tạp của thuật toán có đúng là $O(znm)$ hay không

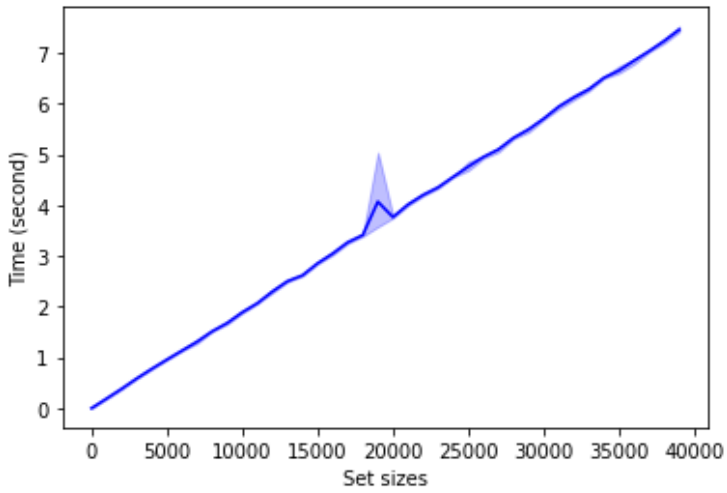
Để kiểm tra độ phức tạp của thuật toán khi tìm trên tập dữ liệu có đúng là $O(znm)$ hay không, ta kiểm tra sự phụ thuộc tuyến tính của thời gian chạy của thuật toán vào $z, n, m$. Vì $z, n, m$ độc lập với nhau nên để kiểm tra sự phụ thuộc tuyến tính, ta giữ cố định 2 trong 3 tham số và đo thời gian chạy của thuật toán với tham số còn lại thay đổi. Ta chia các trường hợp ra làm ba thí nghiệm nhỏ:

- Thí nghiệm 1: Giữ $n, m$ cố định, thay đổi $z$
- Thí nghiệm 2: Giữ $z, m$ cố định, thay đổi $n$
- Thí nghiệm 3: Giữ $z, n$ cố định, thay đổi $m$

### c.i. Thí nghiệm 1: Kiểm tra phụ thuộc tuyến tính vào $z$

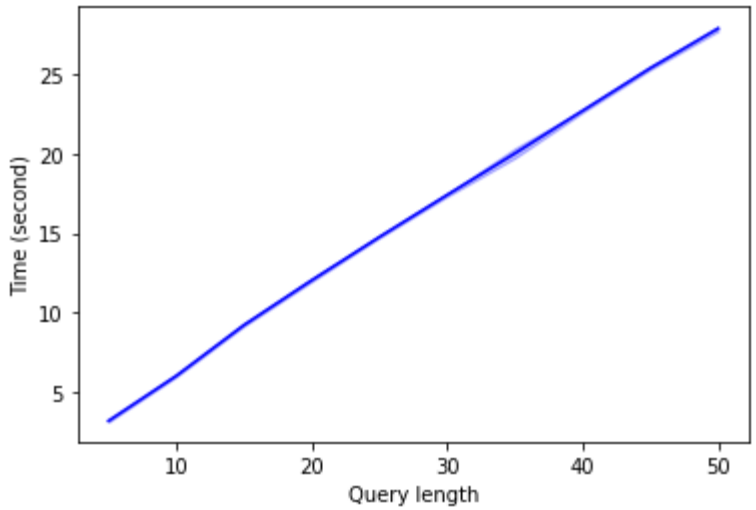
Thí nghiệm 1 được tiến hành như sau: đưa vào thuật toán một chuỗi query $q$ = "ho chi minh" cố định và thay đổi kích thước của tập dữ liệu với $z = 1000, 2000, 3000, ....$ Với mỗi $z_i$ như vậy là chọn ra một tập mẫu ngẫu nhiên $s_i$ có kích thước $z_i$ từ tập dữ liệu ban đầu. Sau đó, ta đo thời gian tìm $q$ trên tập mẫu $s_i$ đó. Ta đo thời gian tìm $q$ trên tập $s_i$ như vậy ba lần và lấy giá trị trung bình.

Biểu đồ dưới đây thể hiện mối quan hệ giữa thời gian chạy và $z$ khi $n, m$ được giữ cố định. Đường nét liền là đường nối các giá trị trung bình ở mỗi lần chạy. Miền màu xanh nhạt bao xung quanh đường nét liền là nối giữa các giá trị lớn nhất và nhỏ nhất ở mỗi lần chạy. Từ biểu đồ có thể thấy thời gian chạy của thuật toán có một sự phụ thuộc tuyến tính vào kích thước của tập dữ liệu.




### c.ii. Thí nghiệm 2: Kiểm tra phụ thuộc tuyến tính vào $n$

Tương tự với thí nghiệm 1, ta giữ cố định $z$ và $m$ bằng cách đưa vào thuật toán toàn bộ dữ liệu gốc. Ta thay đổi $n = 1, 5, 10, 15, \dots 50$. Với mỗi $n_i$, ta tạo ngẫu nhiên chuỗi query $q_i$ có chiều dài là $n_i$ và đo thời gian tìm chuỗi $q_i$ đó. Biểu đồ dưới đây cho thấy thời gian chạy thay đổi thế nào khi $n$ thay đổi.

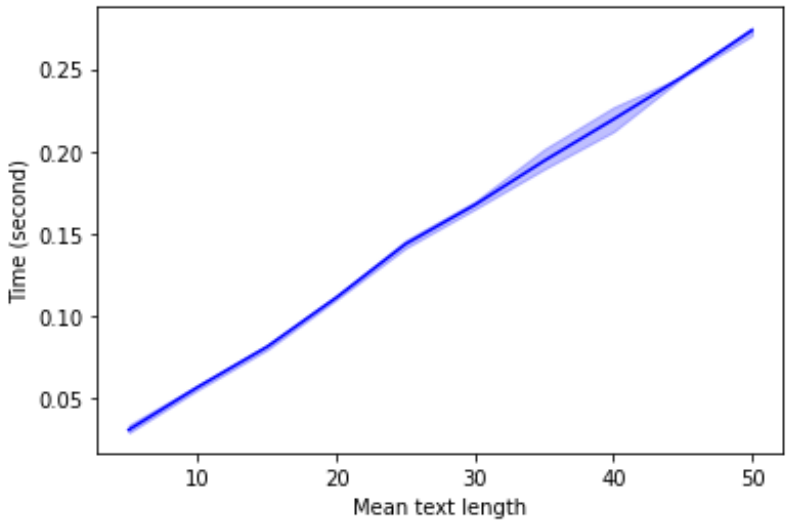



Như vậy, kết quả của thí nghiệm 2 cũng tương tự như khi ta thay đổi $z$. Ta kết luận thời gian chạy của thuật toán phụ thuộc tuyến tính vào $n$.

### c.iii. Thí nghiệm 3: Kiểm tra sự phụ thuộc tuyến tính vào $m$

Do hạn chế về độ dài các chuỗi trong tập dữ liệu thành phố đã sử dụng ở hai thí nghiệm trước gây khó khăn trong việc làm thí nghiệm với $m$ thay đổi, ở thí nghiệm này ta không sử dụng tập dữ liệu đó nữa.

Ta thiết kế thí nghiệm 3 như sau: ta chọn cố định một query $q$ = "ha noi", chọn cố định $z = 500$. Ta làm thí nghiệm với $m = 1, 5, 10, \ldots, 50$. Với mỗi $m_i$ như vậy ta tạo ngẫu nhiên 500 chuỗi ASCII để làm tập dữ liệu. Sau đó ta đo thời gian tìm $q$ trên tập dữ liệu này. Biểu đồ dưới đây thể hiện mối quan hệ giữa thời gian chạy và $m$.

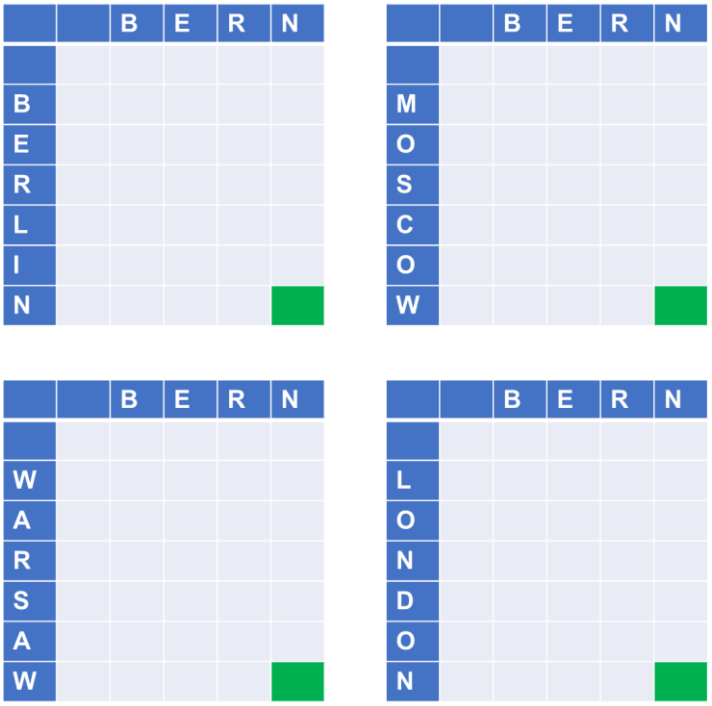



Như vậy có thể thấy thời gian chạy cũng phụ thuộc tuyến tính vào $m$. Do đó dựa trên thực nghiệm ta có thể khẳng định độ phức tạp của thuật toán chính là $O(znm)$.

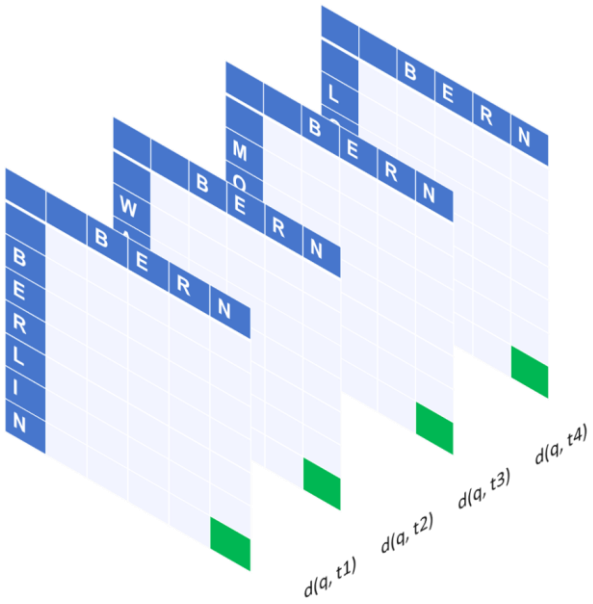
## d. Cải thiện bằng cách sử dụng vectorization

Để việc tìm chuỗi gần giống nhanh hơn, ta có thể tìm khoảng cách từ chuỗi query $q$ đến nhiều chuỗi cùng lúc thay vì phải tìm tuần tự trên từng chuỗi có trong tập dữ liệu. Ta làm điều này bằng cách tận dụng kỹ thuật vectorization.

Hình dưới đây mô tả rõ hơn về ý tưởng này. Giả sử ta muốn tìm khoảng cách của chuỗi $q$ = BERN đến 4 chuỗi $t_1, t_2, t_3, t_4$ lần lượt là BERLIN, MOSCOW, WARSAW và LONDON. Nếu làm theo cách tuần tự, ta cần tạo ra 4 bảng tương ứng với mỗi $t_i$, thực hiện việc điền giá trị vào các ô của từng bảng và lấy ra giá trị ở ô cuối cùng bên phải. Ta thực hiện việc này lần lượt với từng bảng.




Với cách sử dụng vectorization, ta “stack” các bảng này lại thành một ma trận 3 chiều và thực hiện việc cập nhật giá trị trên cả 4 bảng cùng lúc.




Ở đây ta lợi dụng việc các từ BERLIN, MOSCOW, WARSAW và LONDON có cùng độ dài với nhau, do đó bảng quy hoạch động của mỗi từ có cùng kích thước với nhau, hơn nữa cách cập nhật giá trị ở mỗi vị trí trong bảng cũng giống nhau. Sử dụng vectorization thông qua các thư viện như NumPy có thể giúp tăng tốc độ tìm kiếm hơn nhiều so với tìm kiếm một cách tuần tự, vì các thư viện này tối ưu hóa rất tốt các thao tác tính toán trên mảng.

Hàm dưới đây cài đặt việc tính khoảng cách từ chuỗi a có độ dài $m$ đến $k$ chuỗi có cùng độ dài là $n$. $k$ chuỗi cùng độ dài này được gom lại với nhau thành một ma trận kích thước $n \times k$ và được truyền vào bằng tham số b. Ví dụ các chuỗi “berlin”, “moscow”, “warsaw” và “london” được gom lại thành ma trận như sau:

$$b = \begin{bmatrix} \text{b} & \text{m} & \text{w} & \text{l} \\ \text{e} & \text{o} & \text{a} & \text{o} \\ \text{r} & \text{s} & \text{r} & \text{n} \\ \text{l} & \text{c} & \text{s} & \text{d} \\ \text{i} & \text{o} & \text{a} & \text{o} \\ \text{n} & \text{w} & \text{w} & \text{n} \end{bmatrix}$$

Hàm trả về một mảng chứa khoảng cách từ a đến $k$ chuỗi còn lại.

```
def levenshtein_vec(a: np.array, b: np.array):
    m, n, k = len(a), len(b), b.shape[1]
    d = np.zeros((m+1, n+1, k), dtype=np.uint16)

    for i in range(m+1):
        d[i, 0] = i
    for i in range(n+1):
        d[0, i] = i

    for j in range(1, n+1):
        for i in range(1, m+1):
            cost = a[i-1] != b[j-1]
            d[i, j] = np.min([d[i-1, j-1] + cost,
                              d[i, j-1]   + 1,
                              d[i-1, j]   + 1], axis=0)

    return d[m, n]
```

Sử dụng cách vectorization này yêu cầu phải có một bước tiền xử lý đối với tập dữ liệu ban đầu. Ta gom các từ có cùng độ dài trong tập dữ liệu thành từng nhóm và đưa các từ trong từng nhóm thành một ma trận phục vụ cho hàm levenshtein_distance_vec trên.

```
def texts_to_dict(texts):
    len_to_texts = {}
    for word in texts:
        len_to_texts.setdefault(len(word), []).append(word)

    for n, words in len_to_texts.items():
        mat = [[ord(c) for c in word.lower()] for word in words]
        mat = np.asarray(mat).T
        len_to_texts[n] = mat

    return len_to_texts
```

Khi tìm score của một từ, ta thực hiện trên từng nhóm và chọn ra score tốt nhất trong nhóm. Sau cùng, ta chọn ra score tốt nhất ở mọi nhóm.

```
def levenshtein_vec_find(query_text, len_to_texts):
    best_score = -np.inf
    best_result = None

    # Turn the query text into np.array
    text = np.asarray([ord(c) for c in query_text])
```

```
    for length, mat in len_to_texts.items():
        scores = levenshtein_vec(text, mat)

        best_idx = np.argmin(scores)
        score = scores[best_idx]
        score = 1.0 - score / max(len(text), length)

        if score > best_score:
            best_score = score
            best_result = (length, best_idx)

    # Convert best match from integers to string
    best_result = len_to_texts[best_result[0]][:, best_result[1]]
    best_result = ''.join([chr(c) for c in best_result])

    return (best_score, best_result)
```

Kết quả chạy thực nghiệm cho thấy sử dụng vectorization làm tăng tốc đáng kể thời gian tìm kiếm so với khi không sử dụng vectorization, từ mức giây xuống mi-li-giây.

```
%%time
levenshtein_find('ho chiminj', cities)

CPU times: user 6.94 s, sys: 108 ms, total: 7.04 s
Wall time: 6.85 s
(0.5625, 'ho chi minh city')
```

```
%%time
levenshtein_vec_find('ho chiminj', len_to_texts)

CPU times: user 216 ms, sys: 3.32 ms, total: 220 ms
Wall time: 238 ms
(0.5625, 'ho chi minh city')
```

### 4.3. So sánh

Approximate string matching được chia thành hai loại:

- on-line: tập dữ liệu không được xử lý trước, tìm không cần index (fuzzy search, bitap algorithm). Chậm hơn off-line khi xử lý dữ liệu lớn.
- off-line: tập dữ liệu được xử lý trước, sử dụng index để tìm (suffix trees, metric trees, n-gram).

Bảng so sánh độ phức tạp các thuật toán string matching được dẫn từ https://ijrcs.org/wp-content/uploads/201803033.pdf, cùng với một vài thuật toán khác.

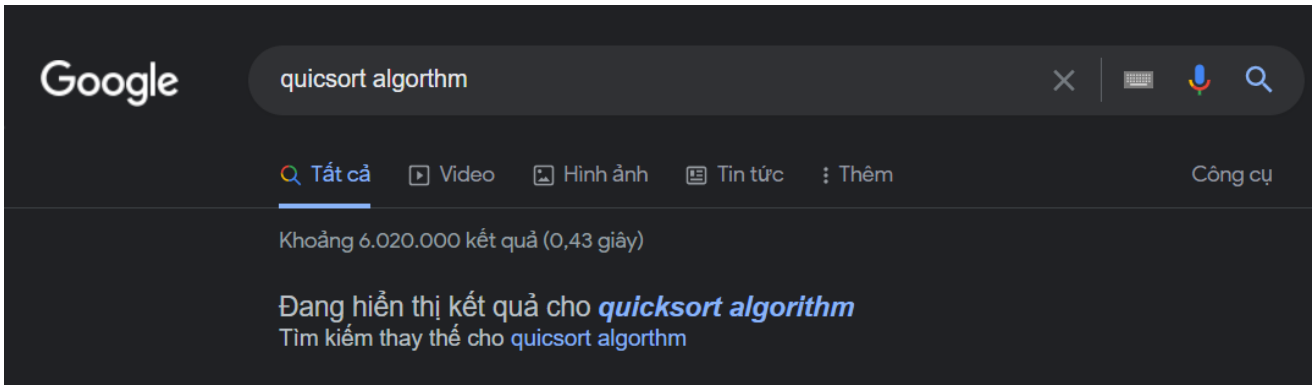
| Algorithm | Preprocessing time | Matching time |
|---|---|---|
| Wagner-Fischer | | $O(znm)$ |
| Rabin–Karp string search algorithm | $\Theta(m)$ | average $\Theta(n + m)$, worst $\Theta((n-m)m)$ |
| Knuth–Morris–Pratt algorithm | $\Theta(m)$ | $\Theta(n)$ |
| Boyer-Moore string search algorithm | $\Theta(m+k)$ | Best - $\Omega\left(\frac{n}{m}\right)$, worst - $O(mn)$ |
| Bitap algorithm | $\Theta(m+k)$ | $O(mn)$ |
| Two-way string-matching algorithm | $\Theta(m)$ | $O(n+m)$ |
| Backward Non-Deterministic Dawg Matching | $O(m)$ | $O(n)$ |
| Backward Oracle Matching | $O(m)$ | $O(n)$ |
| AhoCorasick | | $O(n+m+z)$ |
| CommentZ Walter | | $O(n+m+z)+O(mn)$ |

| | | |
|---|---|---|
| Finite state automation-based search | $O(m)$ | $O(n)$ |
| Suffix tree | Xây dựng: $\Theta(n)$ | Algorithm A: $O(mq + n)$<br>Algorithm B: $O(mq\ log(q) + size\ of\ the\ output)$<br>Algorithm C: $O(m\text{^}2q + size\ of\ the\ output)$<br>https://www.cs.helsinki.fi/u/ukkonen/cpm931.pdf |
| Metric tree (M-tree, vp-trees, cover trees, MVP Trees, BK-trees) | | Tính edit: $O(n + m)$<br>SPIRE02.dvi (unibo.it) |
| n-gram | $O(n \log_2 n)$ (Xây dựng trie) | $O(m \log_2 n)$<br>https://core.ac.uk/download/pdf/82607172.pdf |

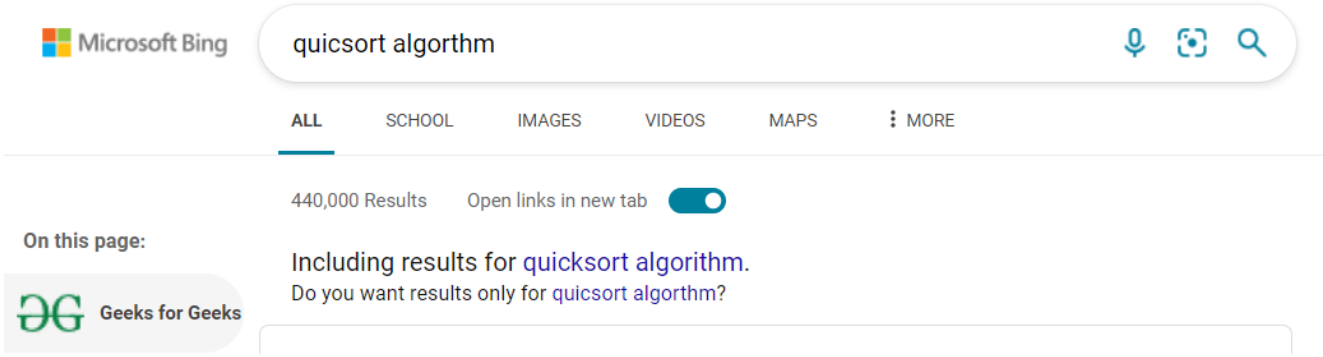
## 5. Ứng dụng

- Thanh tìm kiếm trên các trình duyệt

  Ví dụ khi ta nhập vào một đoạn text bất kỳ có lỗi, như "***quicsort algorthm***" thì kết quả tìm được sẽ có dòng hiển thị một kết quả khác là "***quicksort algorithm***"




*Kết quả trên Google*




*Kết quả trên Microsoft Edge*

- Được tích hợp trong Elasticsearch: là một công cụ tìm kiếm phổ biến dựa trên nền tảng Apache Lucene. Nó cung cấp một bộ máy tìm kiếm dạng phân tán, có đầy đủ công cụ với một giao diện web HTTP có hỗ trợ dữ liệu JSON.
- Các tiện ích hỗ trợ:
    - Flookup: sử dụng fuzzy search để hỗ trợ dọn dẹp dữ liệu trong Google Trang tính. Flookup là lựa chọn hoàn hảo để xử lý các loại tập dữ liệu khác nhau, ngay cả những tập dữ liệu chứa văn bản có phần trùng khớp, lỗi chính tả hoặc các biến thể chính tả.
    - Fuzzy Lookup for Sheets: Tiện ích bổ sung tra cứu mờ trên Google Trang tính này cho phép bạn thực hiện tra cứu mờ và đối sánh mờ trên Google Trang tính cũng như tìm kiếm mờ.

## 6. Tài liệu tham khảo


- https://www.youtube.com/watch?v=SoZ1CVU2DdE
- https://viblo.asia/p/simple-fuzzy-search-BAQ3vV0nMbOr
- https://en.wikipedia.org/wiki/Approximate_string_matching
- https://en.wikipedia.org/wiki/Wagner%E2%80%93Fischer_algorithm
- https://ijrcs.org/wp-content/uploads/201803033.pdf
- https://www.cs.helsinki.fi/u/ukkonen/cpm931.pdf
- http://www-db.disi.unibo.it/research/papers/SPIRE02.pdf
- https://core.ac.uk/download/pdf/82607172.pdf