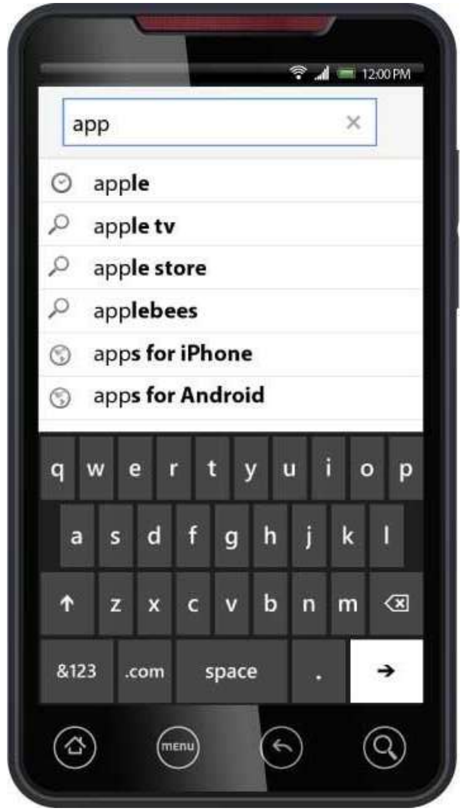
II. Application

1. Remind

- Trie is a data structure that can insert, search, delete a string S in $O(|S|)$.

2. Autocomplete for word

- Autocomplete for word is a feature on most smartphone nowadays.
- Also can be found in search engine, text editor, CLIs, …
- Using trie is one of the earliest way to do this.
- Variations of trie are used to tackle this problem nowadays.



2. Autocomplete for word

Problem

- Given a dictionary of words.
- Each query is a prefix of word, output all words start with this prefix.

Example:

- Input: “ho”
- Output: [‘horse’, ‘house’, ‘hour’, ‘hourglass’, …]

2. Autocomplete for word - Solution

Solution:

- Build a trie for the dictionary.
- With each input, travel to the last node of last character, then do a DFS from this node on trie and output all word.

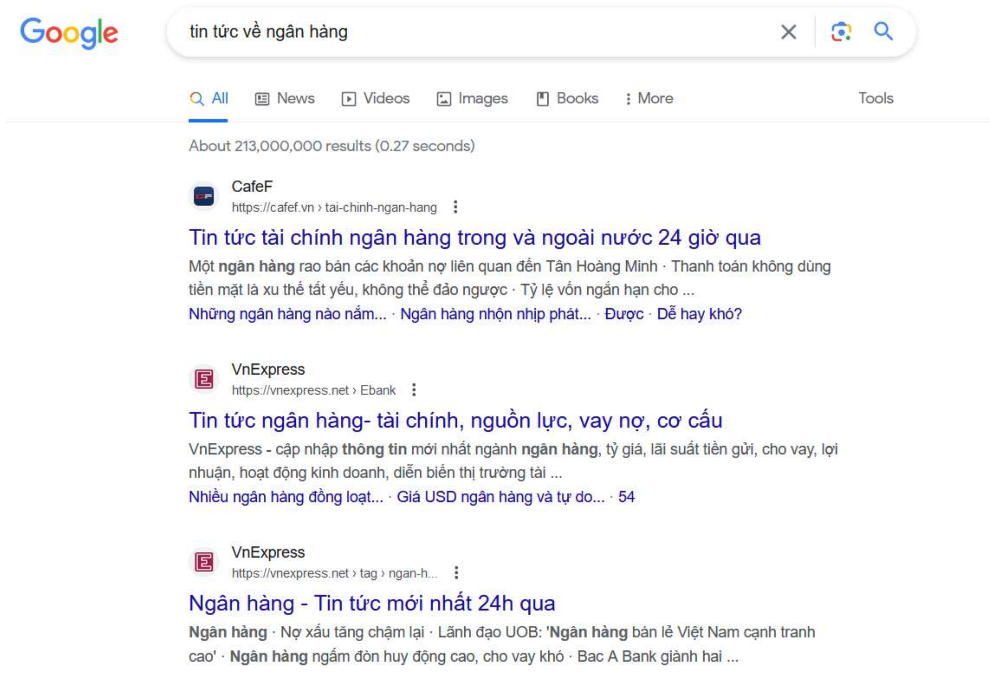
2. Autocomplete for word - Solution

Challenges for the listener:

- Rank output words:.
  - Example: “ho” -> “house” should be standing before “hourglass” in most cases.
  - “Zebra is a ho” -> should recommend “horse” as the first option.
- Update ranking based on user’s behavior.
- Autocorrect for spelling mistake.

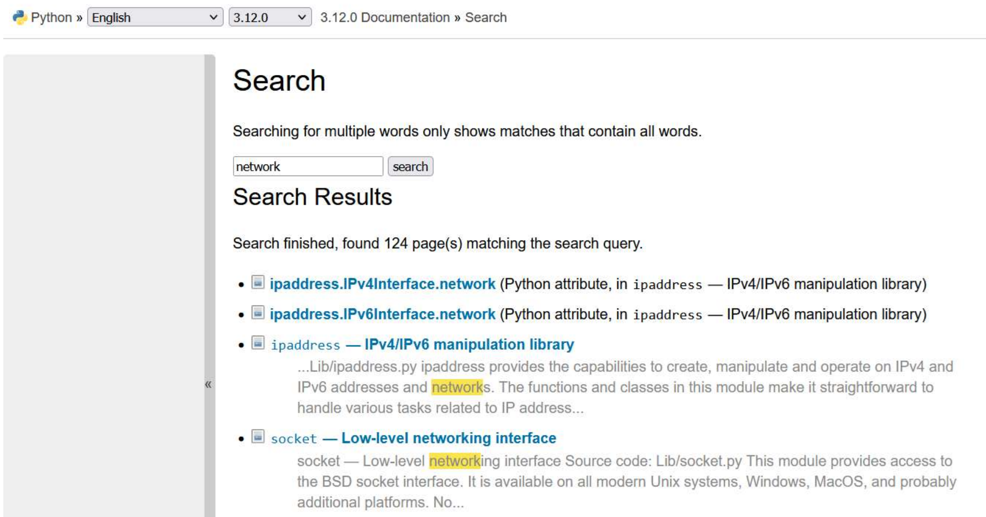
3. Searching document with keywords

- We type keywords into Google, and it returns a bunch of web pages that match those keywords.



3. Searching document with keywords

- We type keywords into the search bar of python documentation, and it returns a bunch of pages that match the keywords.



3. Searching document with keywords

Problem:

- Given a set of document before hand.
- Each query contains some words.
- Output the list of document that has at least one of the words in each query.

Example:

- Input:
  - docs = [[‘nice day’], [‘nice house’], [‘pink house’]]
  - queries = [‘nice’, ‘house’, ‘nice pink’]
- Output: [[0, 1], [1, 2], [0, 1, 2]]

3. Searching document with keywords - Solution

Solution:

- Indexing the set of document by word.
- Add field **doc_indices** which is a set of indices for documents to trie node.
- With each document, with each word, insert that word to the trie and at terminal node, append document index to **doc_indices**.
- With each word in query, we know the **doc_indices** for this word.
- Union all **doc_indices.**

3. Searching document with keywords - Solution

Challenges for the listener:

- Ranking document: which document is more “important” ?
- More query types: AND, OR, EXACT, NOT, …
- Searching a phrase.

III. Exercises

1. Splitting merge words

Problem:

- Given a dictionary and a string written continuously without space.
- Split the string to words.

Example:

- Input: “thisisanexamplestring”
- Output: “this is an example string”

1. Splitting merge words - Solution

Solution:

- Build a trie for the dictionary.
- Let f[i] = True if s[1..i] can be split into words, False otherwise.
- f[0] = True.
- f[i] = True if f[j] = True and s[j+1..i] is a word.
- f[n] = True -> can split.
- Dynamic programming takes O(N.M) with N is the length of string and M is the length of the longest words in dictionary.

1. Splitting merge words - Solution

```
f = [0] * (n+1)
tr = [-1] * (n+1)
f[0] = True
Q = [0]
while len(Q) != 0:
   u = Q.front(); Q.pop()
   for v in [u+1 .. n]:
      if not search(root, s[u+1 .. v]):
         break
      if v not in Q:
         f[v] = True
         tr[v] = u
         Q.push(v)
```

```
u = n
while u != 0:
   words.append(s[tr[u]+1 .. u])
   u = tr[u]

words = reversed(words)
```

2. k-th smallest element

Problem:

- Design a data structure that can supports following operations:
  - Add a number (int32).
  - Find k-th smallest element.
- Every operation should be done online.

Example:

- Input : [‘A 4’, ‘A 8’, ‘A 7’, ‘F 2’, ‘F 1’, ‘A 2’, ‘F 2’]
- Output: [None, None, None, 8, 4, None, 4]

2. k-th smallest element - Solution

Solution:

- Change number to fixed 32-character binary string.
- Each node has count words for each links.
- When insert, add count along the way.
- When query for k-th smallest element, use count to know where to branch.
- Add and Find both have $O(1)$ complexity.

2. k-th smallest element - Solution

```
Node:
  is_end: bool
  next: list[2]
  cnt: list[2] = [0, 0]
```

insert(s)

```
cur = root
for c in s:
  if cur.next[c] == nil
    cur.next[c] = Node()
    cur.cnt[c] += 1
  cur = cur.next[c]
cur.is_end = True
```

search(k)

```
cur = root
res = 0
while k > 0:
  for c in [0, 1]:
    if k > cur.cnt[c]:
      k -= cur.cnt[c]
    else:
      res = res*2 + c
      cur = cur.next[c]
      break
return res
```

3. Lexical sorting

- Problem: Given an array of string, sort them in lexical order.
- Example:
    - Input: [‘c’, ‘b’, ‘a’, ‘an’, ‘bee’, ‘be’, ‘boo’]
    - Output: [‘a’, ‘an’, ‘b’, ‘be’, ‘bee’, ‘boo’, ‘c’]
- Assume N strings, each string is M characters long.
- Using normal sorting algo: O(M.N.logN) (Comparing 2 string takes O(M)).
- Using trie ?

3. Lexical sorting - Solution

- Problem: Given an array of string, sort them in lexical order.
- Example:
  - Input: [‘c’, ‘b’, ‘a’, ‘an’, ‘bee’, ‘be’, ‘boo’]
  - Output: [‘a’, ‘an’, ‘b’, ‘be’, ‘bee’, ‘boo’, ‘c’]
- Assume N strings, each string is M characters long.
- Using normal sorting algo: O(M.N.logN) (Comparing 2 string takes O(M)).
- Using trie ?
  - Build a trie for the array then traveling in pre-order: O(M.N)

Thank you for listening !

Q & A

More References

- VNOI Trie (Vietnamese)
- Autocomplete Algorithms
- More complex tries variation