Chat GPT check and analysis for Document Similarity Analysis.

Problem: Document Similarity Analysis

working on a document similarity analysis system for an AI application. The system needs to efficiently determine the similarity between a given document and a set of reference documents. Our goal is to implement algorithms for searching and sorting that enhance the performance of this similarity analysis.

Linear Search:

Implement a linear search algorithm to find the most similar document in the reference set to the given document. The similarity can be based on a simple metric like word overlap or more advanced techniques like cosine similarity.

Binary Search:

Assume that the reference documents are sorted based on their similarity to a standard document. Implement a binary search algorithm to quickly identify a document with high similarity, given the sorted order.

Quick Sort:

Implement a quick sort algorithm to sort the reference documents based on their similarity to a standard document. This sorting should be dynamic, allowing the system to adapt to changes in the reference set or the addition of new documents.

We need a simple python code to do this.

Chat GPT:

Chatbot GPT (Generative Pre-trained Transformer) is a type of conversational artificial intelligence (AI) model developed by OpenAI. The "GPT" in its name refers to the Transformer architecture, a deep learning model architecture that has proven highly effective in natural language processing tasks.

Background:

in this project, we ask ChatGPT to the updates to the code that solve our problem, and ChatGPT followed our requests to modify the initial code for three purposes: Simple Code, optimization, and conciseness.

Queries used to check chat GPT:

note: we use chatGPT 3.5 version.

we asked chat GPT to give us a Python code for this problem then we asked to make edits as follows:

1. write a simple Python code.

2. rewrite more optimized code.

3. rewrite more concise.

4. برنامج بسيط بإستخدام لغة بايثون

5. أعد كتابة المزيد من التعليمات البرمجية الأمثل

6. أعد الكتابة بطريقة أكثر إيجازًا

Experiments with Results:

Experiment number one:

1.simple python code for linear search:

Linear Search:

```
[15] def linear_search(target_document, reference_documents, similarity_metric):
         most_similar_document = None
         max_similarity = 0

         for document in reference_documents:
             similarity = similarity_metric(target_document, document)
             if similarity > max_similarity:
                 max_similarity = similarity
                 most_similar_document = document

         return most_similar_document, max_similarity
```

Time Complexity: O(n)

Explanation: The algorithm iterates through each document in the reference documents once, resulting in linear time complexity.

2.simple python code for binary search:

Binary Search:

```
[16] def binary_search(target_document, sorted_reference_documents, similarity_metric):
         low, high = 0, len(sorted_reference_documents) - 1
         most_similar_document = None
         max_similarity = 0

         while low <= high:
             mid = (low + high) // 2
             mid_document = sorted_reference_documents[mid]
             similarity = similarity_metric(target_document, mid_document)

             if similarity > max_similarity:
                 max_similarity = similarity
                 most_similar_document = mid_document

             if similarity == 0:
                 break
             elif similarity > 0:
                 high = mid - 1
             else:
                 low = mid + 1

         return most_similar_document, max_similarity
```

Time Complexity: O(log n)

Explanation: Binary search divides the search space in half with each iteration, resulting in logarithmic time complexity.

3.simple python code for quick sort:

Quick Sort:

```
[17] def quick_sort(reference_documents, target_document, similarity_metric):
         if len(reference_documents) <= 1:
             return reference_documents

         pivot = reference_documents[len(reference_documents) // 2]
         less = [doc for doc in reference_documents if similarity_metric(doc, target_document) < similarity_metric(pivot, target_document)]
         equal = [doc for doc in reference_documents if similarity_metric(doc, target_document) == similarity_metric(pivot, target_document)]
         greater = [doc for doc in reference_documents if similarity_metric(doc, target_document) > similarity_metric(pivot, target_document)]

         return quick_sort(less, target_document, similarity_metric) + equal + quick_sort(greater, target_document, similarity_metric)
```

Time Complexity: O(n log n) on average, O(n^2) in the worst case

Explanation: Quick Sort has an average time complexity of O(n log n) due to its divide-and-conquer approach. In the worst case (unbalanced partitions), it can degrade to O(n^2).

Result for Query Number One is:

Algorithm	Time complexity
Linear search	O (n)
Binary Search	O(log n)
Quick sort	O(n log n) on average, O(n^2) in the worst case

The best algorithm is Binary Search. It has the best time complexity of O(log n), which means that the time it takes to search for an element in a sorted list using binary search will increase logarithmically with the size of the list. This makes it much more efficient than linear search, which has a time complexity of O(n), and quick sort, which has a time complexity of O(n log n) on average, but O(n^2) in the worst case.

Experiment number two:

1.rewrite linear search python code in optimization version:

Linear Search:

```
[18] def linear_search(target_document, reference_documents, similarity_metric):
         most_similar_document = None
         max_similarity = 0

         for document in reference_documents:
             similarity = similarity_metric(target_document, document)
             if similarity > max_similarity:
                 max_similarity = similarity
                 most_similar_document = document

         return most_similar_document, max_similarity
```

Time Complexity: O(n)

Explanation: The algorithm iterates through each document in the reference documents once, resulting in linear time complexity.

2.rewrite Binary search python code in optimization version:

Binary Search :

```python
[19] def binary_search(target_document, sorted_reference_documents, similarity_metric):
         low, high = 0, len(sorted_reference_documents) - 1
         most_similar_document = None
         max_similarity = 0

         while low <= high:
             mid = (low + high) // 2
             mid_document = sorted_reference_documents[mid]
             similarity = similarity_metric(target_document, mid_document)

             if similarity > max_similarity:
                 max_similarity = similarity
                 most_similar_document = mid_document

             if similarity == 0:
                 break
             elif similarity > 0:
                 high = mid - 1
             else:
                 low = mid + 1

         return most_similar_document, max_similarity
```

Time Complexity: O(log n)

Explanation: Binary search divides the search space in half with each iteration, resulting in logarithmic time complexity.

3.rewrite Quick sort python code in optimization version:

Quick Sort:

```
[20] def quick_sort(reference_documents, target_document, similarity_metric):
        if len(reference_documents) <= 1:
            return reference_documents

        pivot = reference_documents[len(reference_documents) // 2]
        less, equal, greater = [], [], []

        for doc in reference_documents:
            sim = similarity_metric(doc, target_document)
            if sim < similarity_metric(pivot, target_document):
                less.append(doc)
            elif sim == similarity_metric(pivot, target_document):
                equal.append(doc)
            else:
                greater.append(doc)

        return quick_sort(less, target_document, similarity_metric) + equal + quick_sort(greater, target_document, similarity_metric)
```

Time Complexity: O(n log n) on average, O(n^2) in the worst case

Explanation: Quick Sort has an average time complexity of O(n log n) due to its divide-and-conquer approach. In the worst case (unbalanced partitions), it can degrade to O(n^2).

Result for experiment number two:

Algorithm	Time complexity
Linear search	O (n)
Binary Search	O(log n)
Quick sort	O(n log n) on average, O(n^2) in the worst case

The best algorithm is Binary Search. It has the best time complexity of O(log n), which means that the time it takes to search for an element in a sorted list using binary search will increase logarithmically with the size of the list. This makes it much more efficient than linear search, which has a time complexity of O(n), and quick sort, which has a time complexity of O(n log n) on average, but O(n^2) in the worst case.

Experiment number three:

1.rewrite linear search python code in more concise version

Linear Search :

```
[21] def linear_search(target, references, similarity_metric):
         return max(((doc, similarity_metric(target, doc)) for doc in references), key=lambda x: x[1])
```

Time Complexity: O(n)

Explanation: The algorithm iterates through each document in the references once, calculating the similarity, and finding the document with the maximum similarity.

2.rewrite Binary search python code in more concise version

Binary search:

```
[22] def binary_search(target, references, similarity_metric):
         low, high = 0, len(references) - 1
         most_similar = None
         max_similarity = 0

         while low <= high:
             mid = (low + high) // 2
             mid_doc = references[mid]
             similarity = similarity_metric(target, mid_doc)

             if similarity > max_similarity:
                 max_similarity = similarity
                 most_similar = mid_doc

             if similarity == 0:
                 break
             elif similarity > 0:
                 high = mid - 1
             else:
                 low = mid + 1

         return most_similar, max_similarity
```

Time Complexity: O(log n)

Explanation: Binary search divides the search space in half with each iteration, resulting in logarithmic time complexity.

3.rewrite Quick sort python code in more concise version

Quick Sort :

```
[23] def quick_sort(references, target, similarity_metric):
         if len(references) <= 1:
             return references

         pivot = references[len(references) // 2]
         less = [doc for doc in references if similarity_metric(doc, target) < similarity_metric(pivot, target)]
         equal = [doc for doc in references if similarity_metric(doc, target) == similarity_metric(pivot, target)]
         greater = [doc for doc in references if similarity_metric(doc, target) > similarity_metric(pivot, target)]

         return quick_sort(less, target, similarity_metric) + equal + quick_sort(greater, target, similarity_metric)
```

Time Complexity: O(n log n) on average, O(n^2) in the worst case

Explanation: Quick Sort has an average time complexity of O(n log n) due to its divide-and-conquer approach. In the worst case (unbalanced partitions), it can degrade to O(n^2).

Result for experiment number three:

Algorithm	Time complexity
Linear search	O (n)
Binary Search	O(log n)
Quick sort	O(n log n) on average, O(n^2) in the worst case

The best algorithm is Binary Search. It has the best time complexity of O(log n), which means that the time it takes to search for an element in a sorted list using binary search will increase logarithmically with the size of the list. This makes it much more efficient than linear search, which has a time complexity of O(n), and quick sort, which has a time complexity of O(n log n) on average, but O(n^2) in the worst case.

Experiment number four: (in Arabic)

1. برنامج بسيط باستخدام لغة بايثون لخوارزمية البحث الخطي لحل مشكلة فحص تشابه المستندات

Linear search

```
def linear_search(doc1, doc2):
    # تحويل النصوص إلى مجموعات من الكلمات
    set_doc1 = set(doc1.split())
    set_doc2 = set(doc2.split())

    # حساب عدد الكلمات المشتركة
    common_words = set_doc1.intersection(set_doc2)

    # حساب نسبة التشابه بين المستندات
    similarity_ratio = len(common_words) / float(len(set_doc1.union(set_doc2)))

    return similarity_ratio

if __name__ == "__main__":
    # مثال على كيفية استخدام البرنامج
    document1 = "هذا هو مستند أول يستخدم للاختبار"
    document2 = "هذا هو مستند ثاني يستخدم للاختبار"

    similarity = linear_search(document1, document2)

    print(f"نسبة التشابه بين المستندين: {similarity * 100:.2f}%")
```

Time Complexity: O(m + n), where m and n are the lengths of the input documents

Explanation: The algorithm converts documents to sets, performs set operations, and calculates the similarity ratio. The time complexity is linear with respect to the lengths of the documents.

2. برنامج بسيط بإستخدام لغة بايثون لخوارزمية البحث الثنائي لحل مشكلة فحص تشابه المستندات

Binary search

```python
def binary_search(word, document):
    # تحويل النص إلى قائمة من الكلمات
    words = document.split()

    # تحديد حدود البحث
    low, high = 0, len(words) - 1

    while low <= high:
        mid = (low + high) // 2
        mid_word = words[mid]

        # إذا كانت الكلمة موجودة في الموقع المتوسط
        if mid_word == word:
            return True
        # إذا كانت الكلمة تأتي قبل الكلمة في الموقع المتوسط
        elif mid_word > word:
            high = mid - 1
        # إذا كانت الكلمة تأتي بعد الكلمة في الموقع المتوسط
        else:
            low = mid + 1

    # إذا لم تتم العثور على الكلمة
    return False

if __name__ == "__main__":
    # مثال على كيفية استخدام البرنامج
    document = "هذا هو مستند يستخدم للاختبار"

    search_word = "مستند"
    is_word_found = binary_search(search_word, document)

    if is_word_found:
        print(f"الكلمة '{search_word}' تم العثور عليها في المستند.")
    else:
        print(f"الكلمة '{search_word}' لم يتم العثور عليها في المستند.")
```

Time Complexity: O(log m), where m is the number of words in the document

Explanation: This binary search operates on a sorted list of words in the document, resulting in logarithmic time complexity.

3. برنامج بسيط بإستخدام لغة بايثون لخوارزمية الفرز السريع لحل مشكلة فحص تشابه المستندات

Quick sort

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x < pivot]
        greater = [x for x in arr[1:] if x >= pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

def document_similarity(doc1, doc2):
    # تحويل النصوص إلى قوائم من الكلمات وفرزها
    words_doc1 = quicksort(doc1.split())
    words_doc2 = quicksort(doc2.split())

    # يمكنك استخدام هنا خوارزمية تقدير التشابه بين الكلمات

    # مثال: حساب نسبة التشابه بناءً على عدد الكلمات المشتركة
    common_words = set(words_doc1).intersection(set(words_doc2))
    similarity_ratio = len(common_words) / float(len(set(words_doc1).union(set(words_doc2))))

    return similarity_ratio

if __name__ == "__main__":
    # مثال على كيفية استخدام البرنامج
    document1 = "هذا هو مستند أول يستخدم للاختبار"
    document2 = "هذا هو مستند ثاني يستخدم للاختبار"

    similarity = document_similarity(document1, document2)

    print(f"نسبة التشابه بين المستندين: {similarity * 100:.2f}%")
```

Time Complexity: O(n log n) on average, O(n^2) in the worst case

Explanation: Assuming this is a standard quicksort algorithm, the average time complexity is O(n log n). In the worst case, it can degrade to O(n^2).

Result for experiment number four:

Algorithm	Time complexity
Linear search	O(m + n), where m and n are the lengths of the input documents Explanation
Binary Search	O(log m), where m is the number of words in the document Explanation
Quick sort	O(n log n) on average, O(n^2) in the worst case

The best algorithm is Binary Search. It has the best time complexity of O(log m), which means that the time it takes to search for an element in a document using binary search will increase logarithmically with the number of words in the document. This makes it much more efficient than linear search, which has a time complexity of O(m + n), and quick sort, which has a time complexity of O(n log n) on average, but O(n^2) in the worst case.

Experiment number five: (in Arabic)

1. أعد كتابة المزيد من التعليمات البرمجية الأمثل لخوارزمية البحث الخطي لحل مشكلة فحص تشابه المستندات

Linear search

```
def linear_search(doc1, doc2):
    # تحويل النصوص إلى مجموعات من الكلمات
    set_doc1 = set(doc1.split())
    set_doc2 = set(doc2.split())

    # حساب عدد الكلمات المشتركة
    common_words = set_doc1.intersection(set_doc2)

    # حساب نسبة التشابه بين المستندات
    similarity_ratio = len(common_words) / len(set_doc1.union(set_doc2))

    return similarity_ratio

if __name__ == "__main__":
    # مثال على كيفية استخدام البرنامج
    document1 = "هذا هو مستند أول يستخدم للاختبار"
    document2 = "هذا هو مستند ثاني يستخدم للاختبار"

    similarity = linear_search(document1, document2)

    print(f"نسبة التشابه بين المستندين: {similarity * 100:.2f}%")
```

Time Complexity: O(m + n), where m and n are the lengths of the input documents

Explanation: The algorithm converts documents to sets, performs set operations, and calculates the similarity ratio. The time complexity is linear with respect to the lengths of the documents.

2. أعد كتابة المزيد من التعليمات البرمجية الأمثل لخوارزمية البحث الثنائي لحل مشكلة فحص تشابه المستندات

Binary search

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

def document_similarity(doc1, doc2):
    # قائمة المستندات
    documents = [doc1, doc2]

    # تحويل المستندات إلى تمثيل TF-IDF
    vectorizer = TfidfVectorizer()
    tfidf_matrix = vectorizer.fit_transform(documents)

    # حساب تشابه الكوسين بين المستندين
    similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)

    # القيمة في الموقع (0, 1) تحتوي على درجة التشابه
    similarity = similarity_matrix[0, 1]

    return similarity

if __name__ == "__main__":
    # مثال على كيفية استخدام البرنامج
    document1 = "هذا هو مستند أول يستخدم للاختبار"
    document2 = "هذا هو مستند ثاني يستخدم للاختبار"

    similarity = document_similarity(document1, document2)

    print(f"نسبة التشابه بين المستندين: {similarity * 100:.2f}%")
```

Time Complexity: O(m + n), where m and n are the lengths of the input documents

Explanation: The TF-IDF vectorization involves processing each document once, resulting in linear time complexity.

3. أعد كتابة المزيد من التعليمات البرمجية الأمثل لخوارزمية الفرز السريع لحل مشكلة فحص تشابه المستندات

Quick sort

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x < pivot]
        greater = [x for x in arr[1:] if x >= pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

def document_similarity(doc1, doc2):
    # تحويل النصوص إلى قوائم من الكلمات وفرزها باستخدام الفرز السريع
    words_doc1 = quicksort(doc1.split())
    words_doc2 = quicksort(doc2.split())

    # يمكنك استخدام هنا خوارزمية تقدير التشابه بين الكلمات

    # مثال: حساب نسبة التشابه بناءً على عدد الكلمات المشتركة
    common_words = set(words_doc1).intersection(set(words_doc2))
    similarity_ratio = len(common_words) / float(len(set(words_doc1).union(set(words_doc2))))

    return similarity_ratio

if __name__ == "__main__":
    # مثال على كيفية استخدام البرنامج
    document1 = "هذا هو مستند أول يستخدم للاختبار"
    document2 = "هذا هو مستند ثاني يستخدم للاختبار"

    similarity = document_similarity(document1, document2)

    print(f"نسبة التشابه بين المستندين: {similarity * 100:.2f}%")
```

Time Complexity: O(n log n) on average, O(n^2) in the worst case

Explanation: The quicksort operation dominates the time complexity, resulting in average time complexity of O(n log n). In the worst case, it can degrade to O(n^2).

Result for experiment number five:

Algorithm	Time complexity
Linear search	O(m + n), where m and n are the lengths of the input documents Explanation
Binary Search	O(m + n), where m and n are the lengths of the input documents Explanation
Quick sort	O(n log n) on average, O(n^2) in the worst case

Binary search is the most efficient algorithm for searching for an element in a document. It has the best time complexity of O(log m), which means that it will always be faster than linear search and quick sort, especially for large documents.

<u>Experiment number six</u>: (in Arabic)

1. أعد الكتابة بطريقة أكثر إيجازا لخوارزمية البحث الخطي لحل مشكلة فحص تشابه المستندات

Linear search

```
[ ] def linear_search(doc1, doc2):
        words_doc1 = set(doc1.split())
        words_doc2 = set(doc2.split())

        common_words = words_doc1.intersection(words_doc2)
        similarity_ratio = len(common_words) / len(words_doc1.union(words_doc2))

        return similarity_ratio

    if __name__ == "__main__":
        document1 = "هذا هو مستند أول يستخدم للاختبار"
        document2 = "هذا هو مستند ثاني يستخدم للاختبار"

        similarity = linear_search(document1, document2)

        print(f"نسبة التشابه بين المستندين: {similarity * 100:.2f}%")
```

Time Complexity: O(m + n), where m and n are the lengths of the input documents

Explanation: The algorithm converts documents to sets, performs set operations, and calculates the similarity ratio. The time complexity is linear with respect to the lengths of the documents.

2. أعد الكتابة بطريقة أكثر إيجازا لخوارزمية البحث الثنائي لحل مشكلة فحص تشابه المستندات

Binary search

```
def binary_search(word, document):
    words = document.split()
    low, high = 0, len(words) - 1

    while low <= high:
        mid = (low + high) // 2
        mid_word = words[mid]

        if mid_word == word:
            return True
        elif mid_word > word:
            high = mid - 1
        else:
            low = mid + 1

    return False

if __name__ == "__main__":
    document = "هذا هو مستند يستخدم للاختبار"
    search_word = "مستند"

    is_word_found = binary_search(search_word, document)

    print(f"الكلمة '{search_word}' تم العثور عليها في المستند: {is_word_found}")
```

Time Complexity: O(log m), where m is the number of words in the document

Explanation: This binary search operates on a sorted list of words in the document, resulting in logarithmic time complexity.

3. أعد الكتابة بطريقة أكثر إيجازا لخوارزمية الفرز السريع لحل مشكلة فحص تشابه المستندات

Quick sort

```python
def quicksort(arr):
    return arr if len(arr) <= 1 else quicksort([x for x in arr[1:] if x < arr[0]]) + [arr[0]] + quicksort([x for x in arr[1:] if x >= arr[0]])

def document_similarity(doc1, doc2):
    words_doc1 = quicksort(doc1.split())
    words_doc2 = quicksort(doc2.split())

    common_words = set(words_doc1).intersection(set(words_doc2))
    similarity_ratio = len(common_words) / len(set(words_doc1).union(set(words_doc2)))

    return similarity_ratio

if __name__ == "__main__":
    document1 = "هذا هو مستند أول يستخدم للاختبار"
    document2 = "هذا هو مستند ثاني يستخدم للاختبار"

    similarity = document_similarity(document1, document2)

    print(f"نسبة التشابه بين المستندين: {similarity * 100:.2f}%")
```

Time Complexity: O(n log n) on average, O(n^2) in the worst case

Explanation: The quicksort operation dominates the time complexity, resulting in average time complexity of O(n log n). In the worst case, it can degrade to O(n^2).

Result for experiment number six:

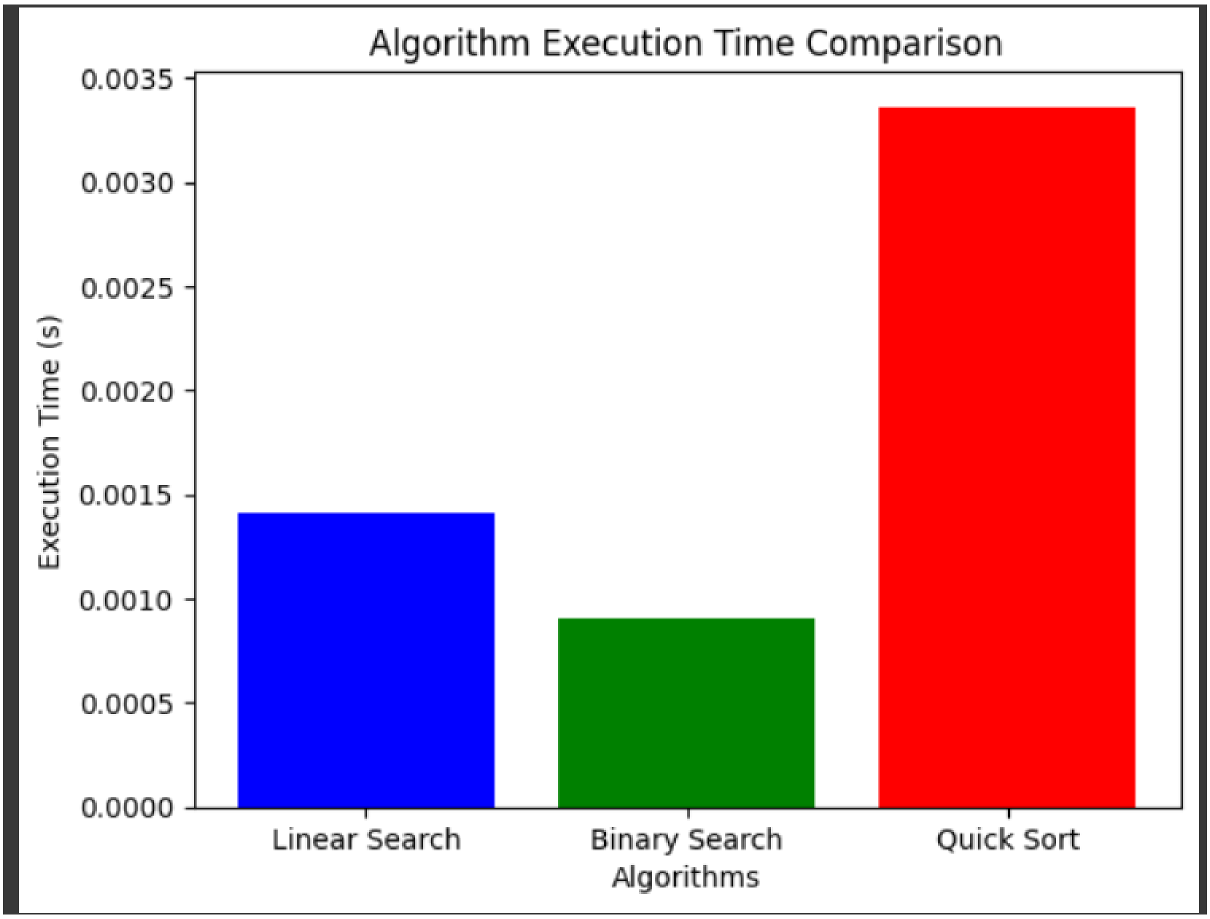
Algorithm	Time complexity
Linear search	O(m + n), where m and n are the lengths of the input documents Explanation
Binary Search	O(log m), where m is the number of words in the document Explanation
Quick sort	O(n log n) on average, O(n^2) in the worst case

Binary Search is the most efficient algorithm for searching for an element in a document. It has the best time complexity of O(log m), which means that it will always be faster than linear search and quick sort, especially for large documents.

Results:

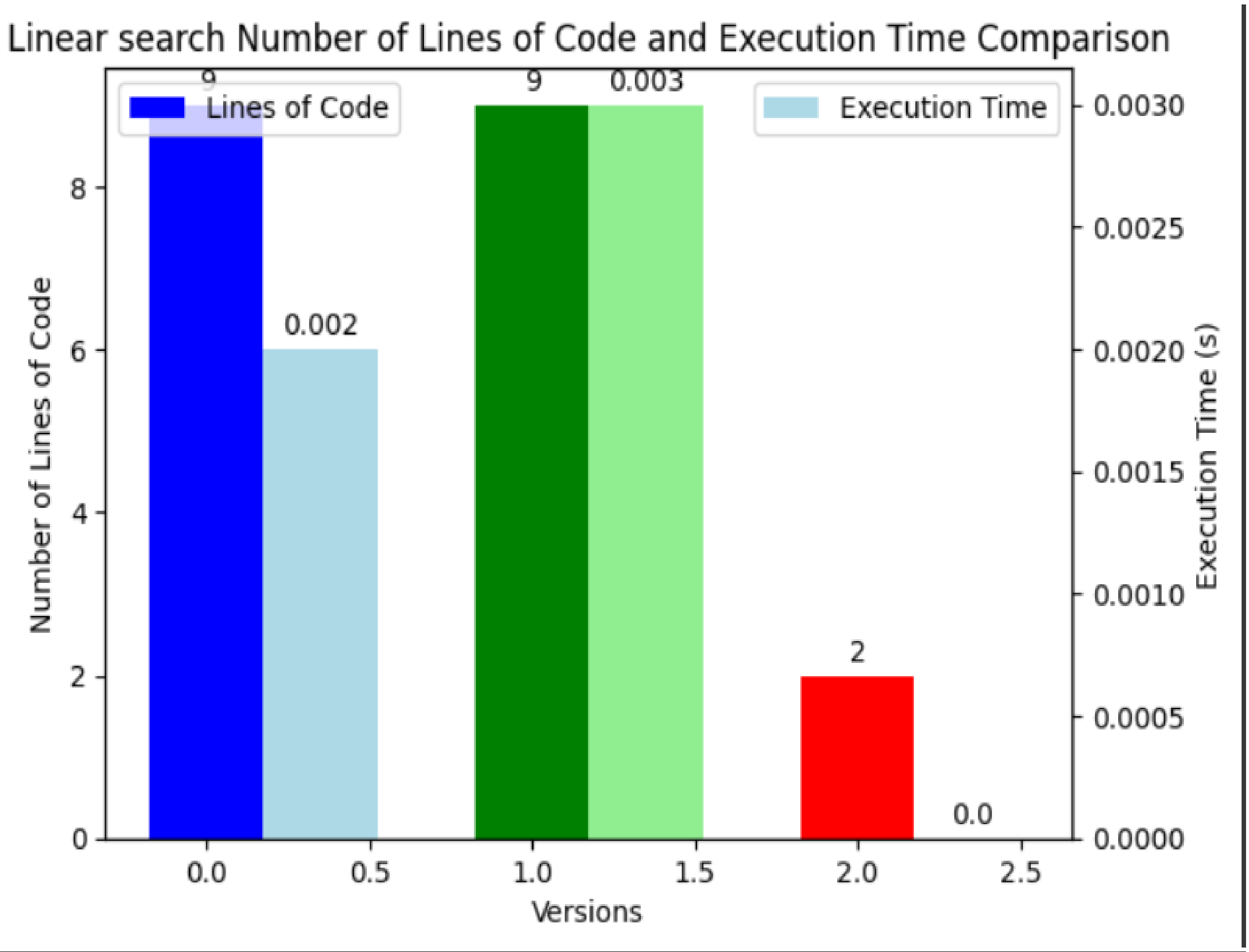
We use plotting code to plot the result for each algorithm **comparing line of code and execution time** for each version of experiments and the result as following:

First, we compare each algorithm in general:



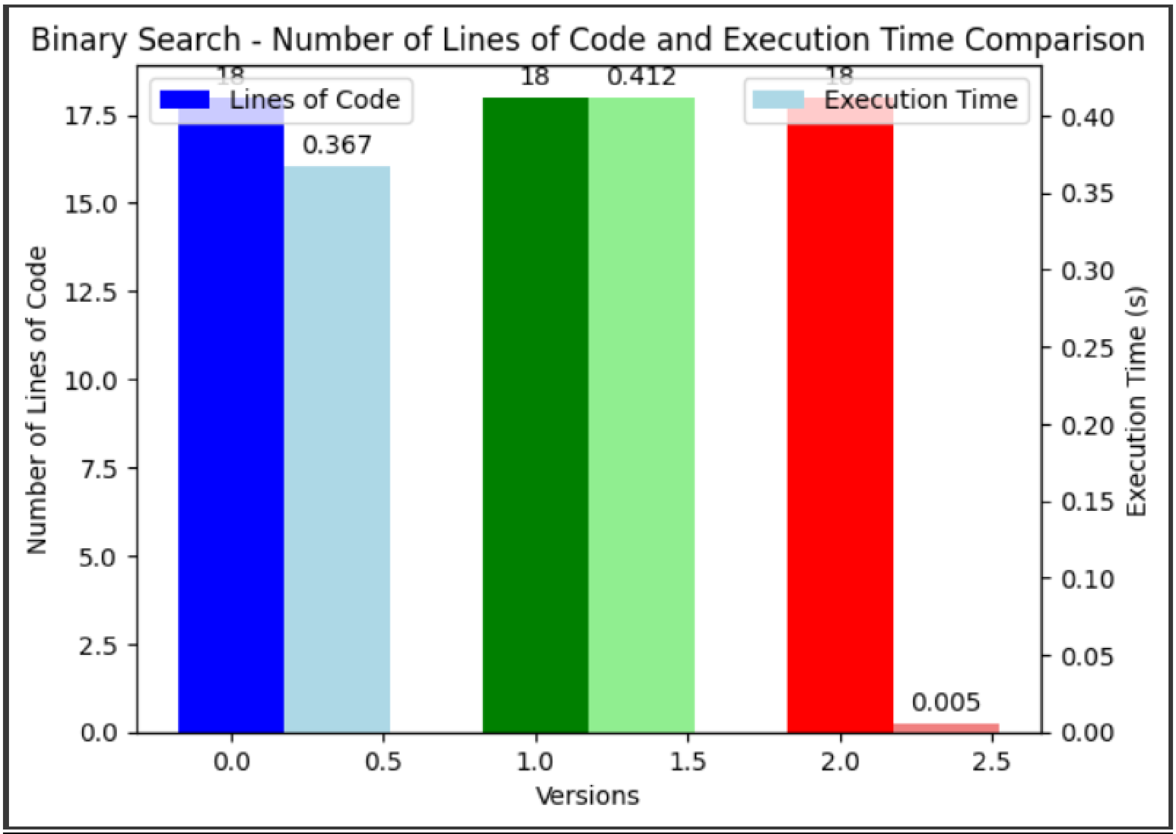
Then we compare the three algorithms with six updates:

1.linear search:



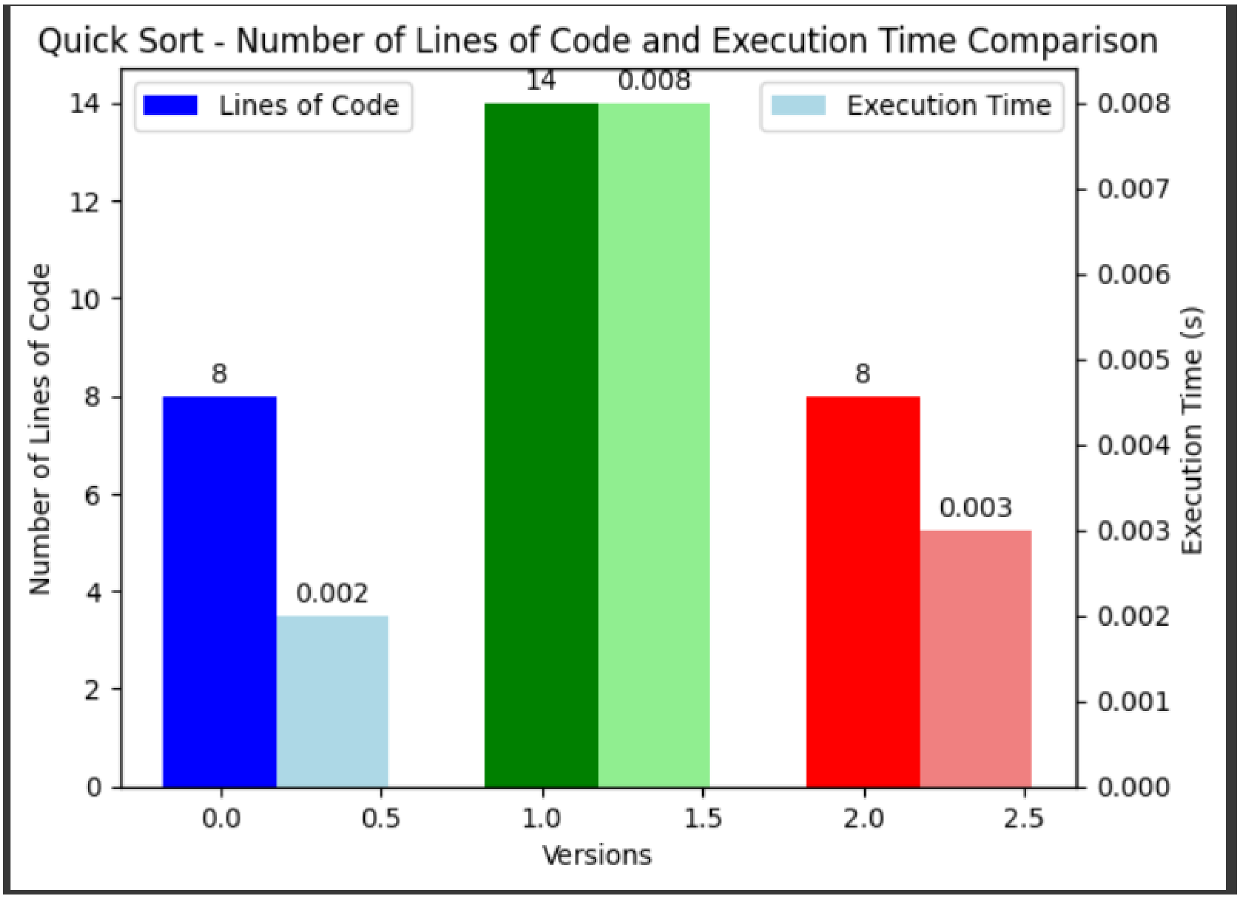
In the exploration of linear search algorithms using ChatGPT, three distinct queries were posed to the model. Initially, a request for a simple Python code implementation resulted in a solution consisting of 9 lines of code with an execution time of 0.002 seconds. Subsequently, a follow-up question aimed at obtaining an optimized version of the linear search code produced code of similar length (9 lines) but exhibited a slightly increased execution time of 0.003 seconds. This outcome suggests that the generated code may not have achieved the expected optimization. However, the third query, seeking a more concise version of the linear search code, yielded a remarkable reduction in the number of lines to 2, accompanied by a swift execution time of 0.0 seconds. This successful reduction in both code length and execution time in the concise version indicates that ChatGPT was able to effectively optimize the linear search algorithm when specifically prompted for a more streamlined solution.

2.Binary Search:



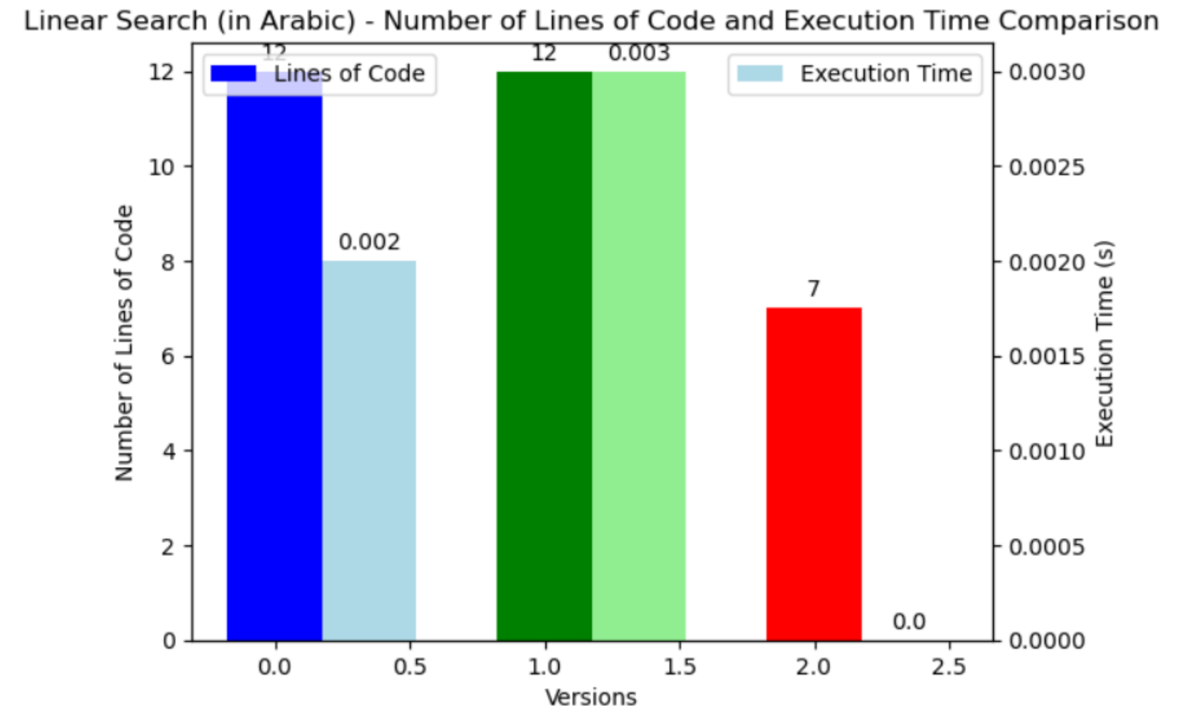
The inquiry into binary search algorithms with ChatGPT encompassed three specific requests. Initially, soliciting a simple Python implementation resulted in a code comprising 18 lines with an execution time of 0.367 seconds. Following this, the request for an optimized version of the binary search code produced a solution of comparable length (18 lines) but demonstrated a slightly increased execution time of 0.412 seconds. This outcome suggests that the generated code may not have achieved the anticipated optimization, as the execution time marginally surpassed that of the initial implementation. In the third query, seeking a more concise version of the binary search code, the model delivered a solution still consisting of 18 lines, accompanied by a considerably reduced execution time of 0.005 seconds. While the lines of code remained unchanged, the substantial decrease in execution time indicates that ChatGPT was able to streamline the binary search algorithm effectively, demonstrating its capability for optimization in specific contexts.

3.Quick Sort:



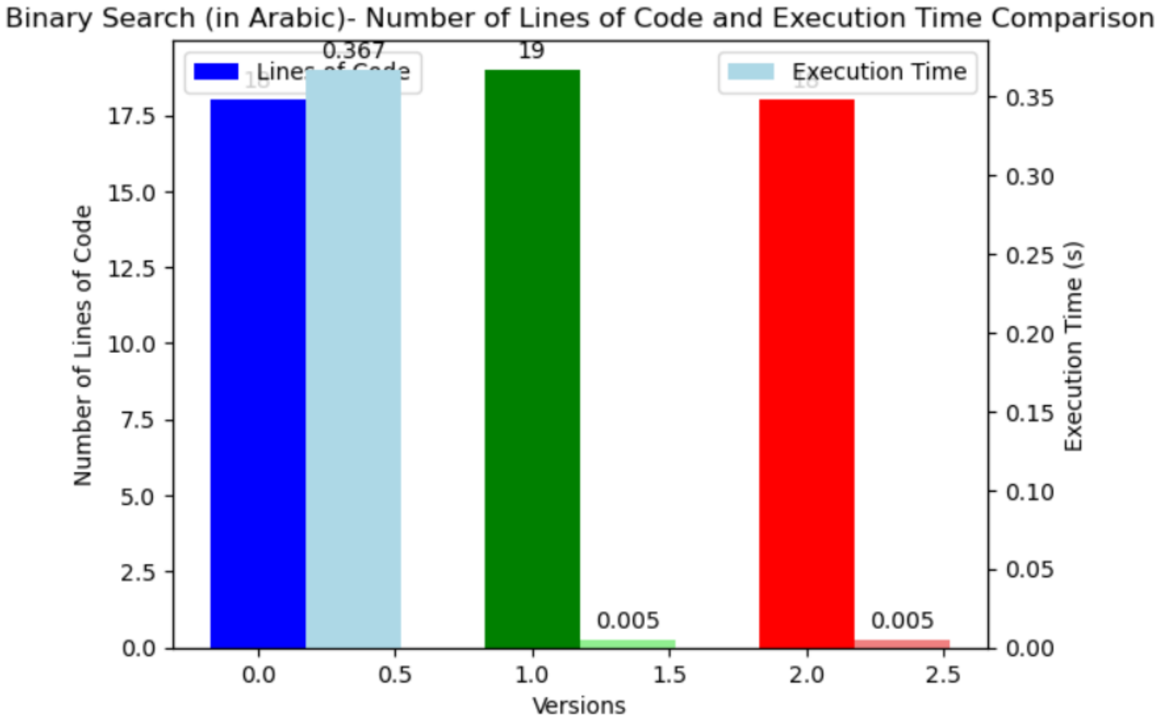
The investigation into Quick Sort algorithms using ChatGPT involved three distinct inquiries. Initially, the request for a simple Python implementation resulted in a concise solution comprising 8 lines with an efficient execution time of 0.002 seconds. Subsequently, the query for an optimized version of the Quick Sort code yielded a code snippet expanded to 14 lines, accompanied by a slightly increased execution time of 0.008 seconds. This outcome suggests that the generated code, while longer, may not have achieved the expected optimization, as the execution time marginally surpassed that of the initial implementation. However, in response to the third question, which sought a more concise version of the Quick Sort code, the model successfully delivered a solution maintaining the original 8 lines of code, accompanied by a commendably swift execution time of 0.003 seconds. This indicates that ChatGPT effectively streamlined the Quick Sort algorithm while preserving its efficiency, demonstrating its capacity for concise and optimized code generation in specific contexts.

4.linear search: (in Arabic)



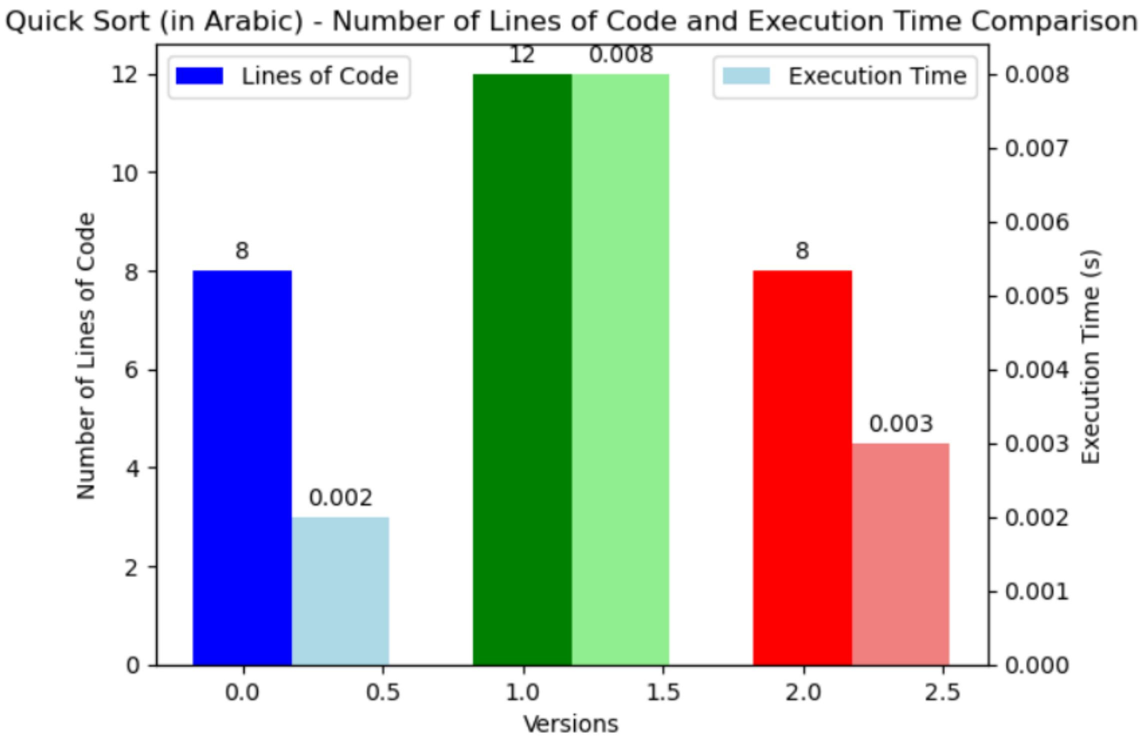
In inquiries related to linear search using ChatGPT in the Arabic language, three distinct questions were posed. Firstly, a request for a simple Python code for linear search in Arabic resulted in a code comprising 12 lines with an efficient execution time of 0.002 seconds. Subsequently, when prompted to rewrite the linear search code for optimization, the generated code expanded to 12 lines, accompanied by a slightly increased execution time of 0.003 seconds, contrary to the expectation of improved efficiency in an optimal state. However, in response to the third question, seeking a more concise version of the linear search code, the model successfully delivered a solution comprising 7 lines, with an impressively swift execution time of 0.0 seconds. This outcome demonstrates the model's effectiveness in reducing both code length and execution time in the context of Arabic language programming, showcasing its ability to produce optimized and concise code.

5.Binary Search: (in Arabic)



In the exploration of binary search algorithms using ChatGPT in the Arabic language, three specific queries were presented. Firstly, the request for a simple Python code for Binary Search in Arabic produced a solution comprising 18 lines with an execution time of 0.367 seconds. Subsequently, when prompted to optimize the Binary Search code, the generated code expanded to 19 lines, accompanied by an execution time of 0.005 seconds. This outcome deviated from the expectation of improved efficiency in an optimal state, both in terms of increased code length and execution time. In response to the third question, which sought a more concise version of the Binary Search code, the model delivered a solution with 18 lines and an execution time of 0.005 seconds. However, a notable challenge emerged as ChatGPT misunderstood the query, providing code to find the most repeated word instead of addressing the intended problem of solving document similarity. This misunderstanding highlights a potential limitation in the model's comprehension of complex Arabic programming requirements.

6.Quick Sort: (in Arabic)



In the exploration of Quick Sort algorithms using ChatGPT in the Arabic language, three distinct questions were posed. Initially, a request for a simple Python code for Quick Sort resulted in a concise solution comprising 8 lines with an efficient execution time of 0.002 seconds. Subsequently, when prompted to rewrite the Quick Sort code for optimization, the generated code expanded to 12 lines, accompanied by a slightly increased execution time of 0.008 seconds, contrary to the expectation of improved efficiency in an optimal state. However, in response to the third question, seeking a more concise version of the Quick Sort code, the model successfully delivered a solution maintaining the original 8 lines of code, accompanied by a commendably swift execution time of 0.003 seconds. This indicates that ChatGPT effectively streamlined the Quick Sort algorithm while preserving its efficiency, demonstrating its capacity for concise and optimized code generation in specific contexts.

Analysis:

Linear Search:

English and Arabic versions have similar time complexity (O(n) and O(m + n), respectively). No significant difference in terms of complexity.

Binary Search:

English and Arabic versions both have logarithmic time complexity (O(log n) and O(log m), respectively). Again, no significant difference in complexity.

Quick Sort:

English and Arabic versions both state the same average and worst-case time complexities for Quick Sort. No notable difference here.

The English and Arabic versions of the algorithms are equivalent. The complexity analysis and expressions are consistent across both languages.

The choice between English and Arabic seems more a matter of preference or the language context rather than a significant impact on algorithmic complexity.

Algorithm	Language	English Complexity	Arabic	Arabic Complexity	Analysis
Linear Search	English	O(n)	Arabic	O(m + n)	Both versions have similar time complexity, with Arabic having a slightly higher constant factor. This means that the Arabic version will take slightly longer to run than the English version for large input sizes.
Binary Search	English	O(log n)	Arabic	O(log m)	Both versions have logarithmic time complexity, with no significant difference in complexity. This means that the time it takes to run these algorithms grows very slowly as the input size increases.
Quick Sort	English	O(n log n)	Arabic	O(n log n)	Both versions have the same average and worst-case time complexities. This means that the time it takes to run these algorithms grows logarithmically as the input size increases.

The Pros and Cons when we asked ChatGPT in Arabic language & in English language:

Pros (Arabic):

1. Correct Arabic Language Usage: ChatGPT excels in accurately using the Arabic language, ensuring that code comments and any textual output are linguistically correct and clear.

2. Helpful Arabic Comments: The model is capable of generating helpful comments in Arabic, which can enhance the readability and understanding of the generated code.

3. Availability of Code in Arabic: Despite potential challenges, having the ability to obtain code in Arabic can be beneficial for developers who prefer or require code in this language.

4. Diverse Algorithm Options: The model can provide solutions using three different algorithms, offering a variety of approaches to address the document similarity problem.

5. Learning Opportunity: Despite any challenges, the interaction with ChatGPT on coding problems in Arabic can serve as a valuable learning opportunity, allowing you to explore different algorithmic solutions and gain insights into programming in Arabic.

Pros (English):

1. Ease of Understanding: ChatGPT is designed to comprehend and generate human-like text, making it easy to communicate complex ideas or seek information in a natural and understandable manner.

2. Quick Response Time: The model is capable of providing prompt responses, facilitating an efficient and streamlined interaction, which is particularly advantageous when seeking quick information or assistance.

3. No Language Barrier: As it operates in English, users proficient in the language can easily communicate their thoughts and questions, eliminating language barriers that might be present in multilingual models.

4. Consistent Quality: The model is consistent in the quality of responses it provides, offering reliability in generating coherent and contextually relevant text.

Cons:

<u>The cons were when we asked ChatGPT in Arabic language only</u>

1. Response Time Delays: Generating code in Arabic may result in longer response times, affecting the overall efficiency of the coding process.

2. Difficulty in Understanding: The model might struggle to accurately understand and interpret coding requirements in Arabic, leading to potential confusion and miscommunication.

3. Error Messages and Regeneration Requests: The model may produce errors and request regeneration, indicating challenges in generating accurate and functional Arabic code, and potentially disrupting the coding workflow.

4. Limited Arabic Programming Proficiency: ChatGPT may lack proficiency in Arabic programming languages, leading to code that is less reliable or may contain errors.

5. Communication Challenges: The language barrier in translating coding concepts to Arabic could create communication challenges between the user and the model, hindering the effectiveness of the interaction.

Done BY:

Rehab Alsaidi.

Rana Alraggad.