Instituto Politécnico Nacional

Escuela Superior de Computo

Unidad de académica:

Análisis y Diseño de algoritmos

Práctica de laboratorio 2:

“Implementación y Evaluación de Algoritmos de Ordenamiento y Búsqueda”

Equipo:

Solís Lugo Mayra

Solares Velasco Arturo Misael

Veruete Hernández Bryan David

Grupo: 3CV1

Profesor: García Floriano Andrés

Fecha:

18 marzo 2024

Reporte de Práctica: Implementación y Evaluación de Algoritmos de Ordenamiento y Búsqueda

Introducción

Los algoritmos de ordenamiento y búsqueda son elementos fundamentales en el ámbito de la informática y la ciencia de la computación. Estos algoritmos desempeñan un papel crucial en una amplia variedad de aplicaciones, desde la gestión de bases de datos hasta la optimización de algoritmos de búsqueda en internet. En esta práctica, se implementaron varios algoritmos de ordenamiento y búsqueda en el lenguaje de programación Python, y se evaluaron sus rendimientos mediante pruebas empíricas con diferentes conjuntos de datos.

Algoritmos Implementados

Algoritmos de Ordenamiento

Mergesort

El algoritmo Mergesort es un método de ordenamiento basado en la técnica de dividir y conquistar. Divide repetidamente la lista en subconjuntos más pequeños, los ordena y luego los fusiona para obtener la lista ordenada completa. La complejidad temporal del algoritmo Mergesort es $O(nlogn)$ en el peor de los casos.

Quicksort

Quicksort es otro algoritmo de ordenamiento que sigue el enfoque de dividir y conquistar. Selecciona un elemento pivote de la lista y reorganiza los elementos de la lista de manera que los elementos menores que el pivote estén a su izquierda y los elementos mayores estén a su derecha. Luego, el algoritmo se aplica recursivamente a las sublistas generadas. La complejidad temporal promedio de Quicksort es $O(n \log n)$ aunque puede degradarse a $O(n^2)$ en el peor de los casos.

```python
# Implementación de Quicksort
def quick_sort(arr):
    if len(arr) ⩽ 1:
        return arr
    else:
        pivot = arr[0]
        less_than_pivot = [x for x in arr[1:] if x ⩽ pivot]
        greater_than_pivot = [x for x in arr[1:] if x > pivot]
        return quick_sort(less_than_pivot) + [pivot] + quick_sort(greater_than_pivot)
```

Algoritmos de Búsqueda

Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada. Divide repetidamente el espacio de búsqueda a la mitad, eliminando la mitad del espacio en cada iteración hasta que el elemento deseado se encuentre. La complejidad temporal de la búsqueda binaria es $O(log\ n)$, lo que la hace especialmente eficiente para grandes conjuntos de datos.

```python
# Implementación de búsqueda binaria
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low ⩽ high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Búsqueda Ternaria

La búsqueda ternaria es una variante de la búsqueda binaria que divide el espacio de búsqueda en tres partes en lugar de dos. Si bien sigue siendo eficiente, su complejidad temporal es ligeramente mayor que la búsqueda binaria, con una complejidad de $O(log_3\ n)$.

```python
# Implementación de búsqueda ternaria
def ternary_search(arr, target):
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3
        if arr[mid1] == target:
            return mid1
        elif arr[mid2] == target:
            return mid2
        elif target < arr[mid1]:
            right = mid1 - 1
        elif target > arr[mid2]:
            left = mid2 + 1
        else:
            left = mid1 + 1
            right = mid2 - 1
    return -1
```

Implementación y Evaluación

Los algoritmos mencionados fueron implementados en Python en el archivo algoritmos.py. Además, se desarrolló el archivo logica_resultados.py para realizar pruebas empíricas y evaluar el rendimiento de los algoritmos. Este archivo contiene funciones para medir el tiempo de ejecución de cada algoritmo, así como para generar resultados en formato de texto y gráficos.

El archivo main.py es el punto de entrada del programa, donde se definen las tareas principales a ejecutar. Estas incluyen la generación de arreglos de números aleatorios, la ejecución de pruebas empíricas con diferentes tamaños de conjuntos de datos y la generación de archivos CSV y gráficos de los resultados.

Resultados y Análisis

Se realizaron pruebas con conjuntos de datos de diferentes tamaños para evaluar el rendimiento de los algoritmos implementados. Estos resultados fueron presentados tanto en formato de texto en la línea de comandos como en gráficos generados automáticamente.

Los resultados mostraron que, en general, Mergesort y Quicksort tuvieron tiempos de ejecución similares en conjuntos de datos de tamaño pequeño a moderado. Sin embargo, Mergesort mostró una mayor consistencia en el tiempo de ejecución a medida que el tamaño del conjunto de datos aumentaba. La búsqueda binaria demostró ser altamente eficiente, con tiempos de ejecución muy bajos y consistentes en todos los tamaños de conjunto de datos probados. Por otro lado, la búsqueda ternaria, aunque eficiente, mostró tiempos de ejecución ligeramente más altos en comparación con la búsqueda binaria.

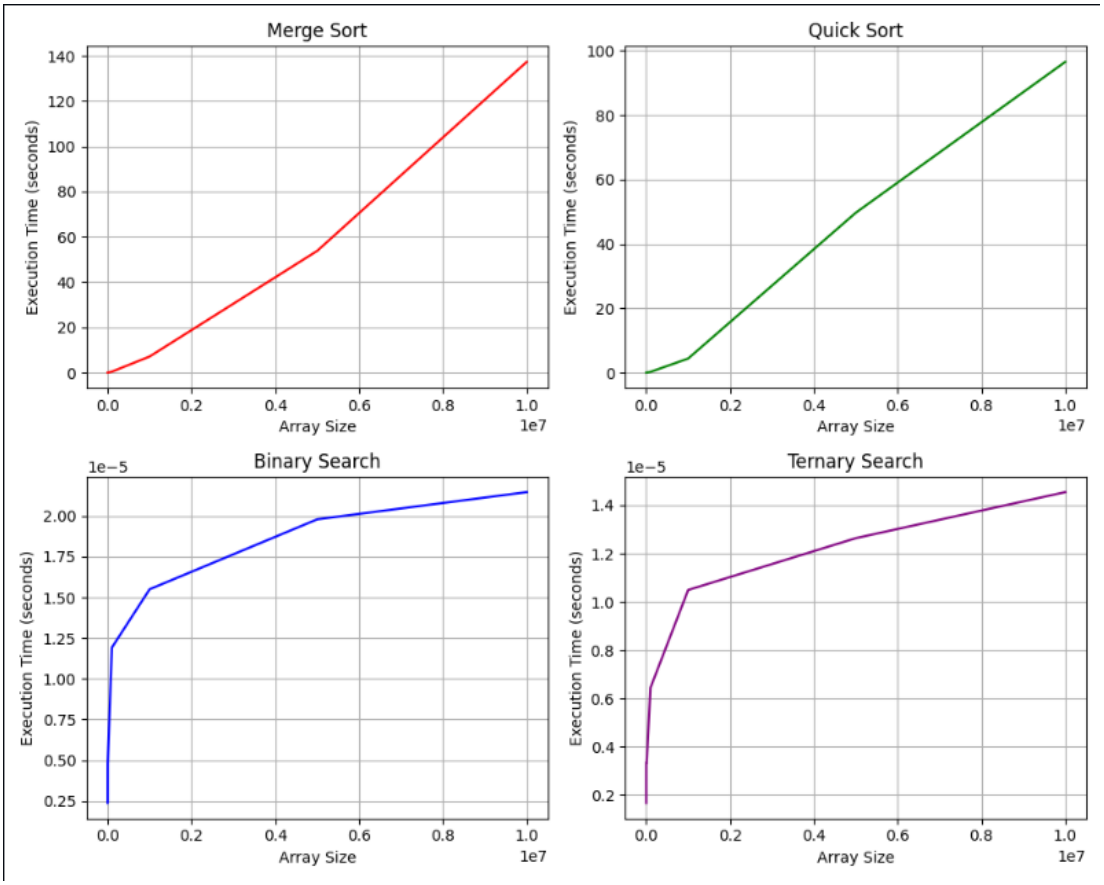

Ilustración 1 Resultados gráficos de los tiempos de ejecución en n tamaños de datos.

Conclusiones

En conclusión, la elección del algoritmo adecuado depende de varios factores, como el tamaño y la naturaleza de los datos, así como los requisitos de rendimiento específicos de la aplicación. Los algoritmos de ordenamiento y búsqueda implementados en esta práctica proporcionan herramientas poderosas para manejar y procesar datos de manera eficiente. Es fundamental comprender las características y complejidades de cada algoritmo para tomar decisiones informadas al diseñar sistemas y aplicaciones que manejen grandes cantidades de datos.