Universidad Nacional de San Antonio Abad del Cusco

Escuela Profesional de Ingeniería Informática y de Sistemas

“Algoritmos de ordenamiento en lenguaje ensamblador y lenguajes de alto nivel.”

Lenguaje Ensamblador

Estudiantes:

200820 - Atauchi Mamani Jose Emilio
200826 - Rodriguez Phillco Jaime Antonio

Docente:

Edwin Carrasco Poblete

Semestre:

2024-II

I. Presentación

Los estudiantes del curso de Lenguaje Ensamblador se complacen en presentar el trabajo comparación de algoritmos de ordenamiento en lenguaje Ensamblador y lenguaje de alto nivel**.** El objetivo de este proyecto es comparar el tiempo de ejecución de algoritmos de ordenamiento implementados tanto en lenguaje ensamblador como en un lenguaje de alto nivel, proporcionando así una visión clara de las diferencias de rendimiento entre ambas implementaciones.

II. Introducción

A lo largo de los años, los componentes de las computadoras han mejorado notablemente, permitiendo la ejecución de programas a mayores velocidades y con mayor eficiencia. A pesar de estos avances, la elección del lenguaje de programación y su relación con el hardware sigue siendo un factor crucial para el rendimiento de los sistemas.

Este trabajo explora la comparación entre algoritmos de ordenamiento implementados en lenguaje ensamblador y en un lenguaje de alto nivel, con el objetivo de evaluar las diferencias en el tiempo de ejecución entre ambas implementaciones. A través de esta comparación, se busca ver la importancia del lenguaje ensamblador en el rendimiento de los algoritmos.

III. Índice General

IV. Índice de Figuras

V. Índice de Tablas

No se encuentran elementos de tabla de ilustraciones.

VI. Marco Teórico

A. BubbleSort

1. Descripción del algoritmo

2. Complejidad temporal y de memoria

Asddsa

2.1. Complejidad temporal

2.2. Complejidad de memoria

3. Pseudocódigo

4. Ejemplo

B. QuickSort

1. Descripción del algoritmo

Quicksort aplica el método de divide y vencerás. A continuación, se describe el proceso de tres pasos de divide y vencerás para ordenar un subarreglo A[p : r]

a. **Dividir**: Se particiona el subarreglo A[p : r] en dos subarreglos (posiblemente vacíos), A[p : q - 1] (el lado bajo) y A[q + 1 : r] (el lado alto). Esto se realiza de tal forma que cada elemento en el lado bajo de la partición es menor o igual al pivote A[q], el cual, a su vez, es menor que cada elemento en el lado alto. Como parte de este procedimiento, se calcula el índice q del pivote.
b. **Vencer**: Se aplica recursivamente el algoritmo de Quicksort para ordenar cada uno de los subarreglos A[p : q - 1] y A[q + 1 : r].
c. **Combinar**: No es necesario realizar ninguna operación adicional para combinar, ya que los dos subarreglos ya están ordenados. Todos los elementos en A[p : q - 1] están ordenados y son menores o iguales a A[q] y todos los elementos en A[q + 1 : r] están ordenados y son mayores que el pivote A[q].

1.1. Particionamiento del Arreglo

La clave del algoritmo es el procedimiento de PARTITION, que reorganiza el subarreglo A[p : r] en su lugar, devolviendo el índice del punto divisorio entre los dos lados de la partición.

Proceso de particionamiento:

- **Inicialización**: Antes de la primera iteración, se establece i = p – 1 y j = p. Dado que no hay valores entre p e i, ni entre i + 1 y j - 1, las dos primeras condiciones se cumplen automáticamente. La asignación en la línea 1 asegura que la tercera condición se mantenga.
- **Mantenimiento**: Durante cada iteración, el algoritmo verifica si A[j] > x. Si es cierto, simplemente incrementa j. Si A[j] ≤ x, el algoritmo incrementa i, intercambia A[i] y A[j], y luego incrementa j. Este intercambio asegura que A[i] ≤ x y A[j - 1] > x, manteniendo el invariante de bucle.

- **Terminación**: El bucle termina después de r - p iteraciones, momento en el cual el subarreglo no examinado A[j : r - 1] está vacío. Todos los elementos del arreglo pertenecen a una de las tres regiones descritas por el invariante. Así, los valores del arreglo se dividen en tres conjuntos: elementos menores o iguales a x, elementos mayores que x, y el pivote en sí x.

Finalmente, las dos últimas líneas de **PARTITION** intercambian el pivote con el primer elemento mayor que xxx, colocándolo en su posición correcta en el arreglo, y luego devuelven el nuevo índice del pivote. Esto garantiza que, después de la línea 3 de QUICKSORT, A[q] es estrictamente menor que cada elemento de A[q + 1 : r].

2. Complejidad temporal y de memoria

2.1. Complejidad temporal

La eficiencia de Quicksort varía según cómo se seleccionen los pivotes y cómo estén distribuidos los elementos en el arreglo. Las complejidades temporales en diferentes escenarios son:

- **Caso Promedio O(nlogn)**: En la mayoría de los casos, si el pivote divide el arreglo en partes aproximadamente iguales, el rendimiento promedio es O(nlogn). Esto se debe a que se realizan cerca de logn particiones, con n elementos procesados en cada nivel de recursión.
- **Mejor Caso O(nlogn)**: Quicksort alcanza su mejor rendimiento cuando cada partición divide el arreglo en dos mitades iguales. Este caso se da si el pivote es el elemento mediano o cercano al mediano, logrando así una complejidad de O(nlogn).
- **Peor Caso O(n^2)**: El peor caso ocurre cuando el pivote es siempre el elemento más grande o más pequeño, resultando en particiones muy desbalanceadas. Este escenario suele presentarse cuando el arreglo está ya ordenado o casi ordenado, y no se aplican estrategias para elegir un pivote equilibrado. En tales casos, Quicksort tiene una complejidad temporal de O(n^2), ya que cada partición sólo reduce el tamaño del problema en un elemento.

2.2. Complejidad de memoria

Quicksort es un algoritmo in-place, lo que significa que realiza la ordenación directamente en el arreglo, sin necesidad de espacio adicional significativo. La complejidad de memoria depende principalmente de la profundidad de la recursión:

- **Caso Promedio y Mejor Caso O(logn)**: En estos casos, la profundidad de la pila de llamadas recursivas es aproximadamente logn. Por lo tanto, el espacio adicional requerido es O(logn), utilizado principalmente para almacenar las variables de cada llamada recursiva.
- **Peor Caso O(n)**: En el peor de los casos, cuando las particiones están desbalanceadas, la profundidad de la recursión puede llegar a O(n), lo que implica un consumo de memoria lineal. Esto ocurre porque cada llamada recursiva solo reduce el problema en un elemento.

3. Pseudocódigo

Ilustración 1 *Pseudocódigo del algoritmo Quicksort*

```
QUICKSORT(A, p, r)
1  if p < r
2       // Partition the subarray around the pivot, which ends up in A[q].
3       q = PARTITION(A, p, r)
4       QUICKSORT(A, p, q − 1)  // recursively sort the low side
5       QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

(Cormen, Leiserson, Rivest & Stein, 2022, p. 183, cap. 7)

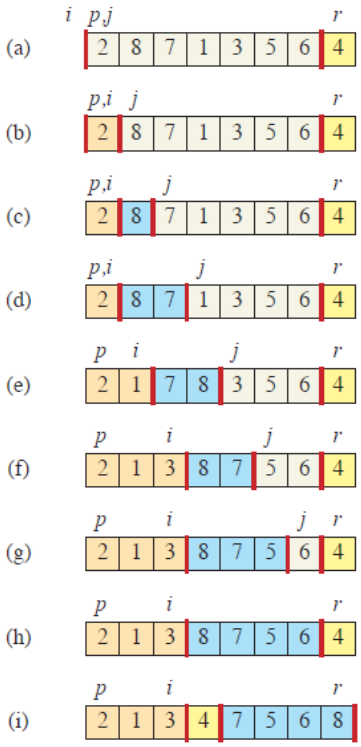
Ilustración 2 *Pseudocódigo del algoritmo Partition*

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4       if A[j] ≤ x                  // does this element belong on the low side?
5            i = i + 1                    // index of a new slot in the low side
6            exchange A[i] with A[j]      // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

(Cormen, Leiserson, Rivest & Stein, 2022, p. 184, cap. 7)

4. Ejemplo

Ilustración 3 *Ejemplo del algoritmo Quicksort*



(Cormen, Leiserson, Rivest & Stein, 2022, p. 185, cap. 7)

El elemento A[r] del arreglo se convierte en el elemento pivote x. Los elementos de color beige pertenecen al lado bajo de la partición, con valores como máximo x. Los elementos azules pertenecen al lado alto, con valores mayores que x. Los elementos en blanco aún no se han asignado a un lado de la partición, y el elemento amarillo es el pivote x.

(a) El arreglo inicial y las variables. Ningún elemento ha sido asignado aún a un lado de la partición.
(b) El valor 2 se "intercambia consigo mismo" y se coloca en el lado bajo.
(c)–(d) Los valores 8 y 7 se colocan en el lado alto.
(e) Los valores 1 y 8 se intercambian, y el lado bajo crece.
(f) Los valores 3 y 7 se intercambian, y el lado bajo crece. (g)–(h) El lado alto de la partición se amplía para incluir 5 y 6, y el bucle termina. (i) La línea 7 intercambia el pivote para que se sitúe entre los dos lados de la partición, y la línea 8 devuelve el nuevo índice del pivote.

VII. Código fuente de los programas de ordenamiento

VIII. Documentación de las pruebas

IX. Resultados e interpretación de las pruebas

X. Referencias