Universidad de los Andes > FACULTAD DE INGENIERÍA Y CIENCIAS APLICADAS

Informe Estructura de Datos y Algoritmos

Profesor:
Jose M. Saavedra

Integrantes:
Benjamis Figueroa
Stefano Romanini

13 de septiembre 2022

Índice:

1. Resumen:

En este informe se verá la importancia del diseño de los algoritmos de ordenamiento a través de una secuencia de experimentos. Primero se analizará la eficiencia de cada algoritmo, Inserción, Selección, Quicksort y Mergesort, calculando el tiempo promedio en ordenar una lista con números flotantes de tamaño n.

Luego de recopilar los datos del primer experimento, se compara el mejor algoritmo del primer experimento con el algoritmo Radix Sort. En esta segunda parte se ejecutarán los dos algoritmos en una lista de números enteros y no flotantes.

Para lograr obtener una mejor precisión en los resultados, se ejecutarán los algoritmos en una misma lista con el fin de comparar eficiencia en el mismo ambiente. Además, se correrán 5 veces los algoritmos para obtener un promedio en su tiempo de ejecución.

Para los experimentos, se utilizarán una cantidad de datos pequeña y una mayor. Las cantidades pequeñas van de 10.000 datos hasta 100.000. Al momento de trabajar con mayores cantidades se trabajará desde 100.000 a 1.000.000 de datos.

Los objetivos de este experimento son lograr implementar cada algoritmo para ordenar una lista de datos con tamaño indefinido. También se recopilaran los datos necesarios para determinar qué algoritmos es más eficiente.

2. Inserción:

Se destaca este algoritmo en ser intuitivo, ya que sigue la estrategia de ordenación de barajas, es decir que vamos a tomar cada elemento del arreglo y lo vamos ordenando de acuerdo uno avanza.

Para implementar lo anterior vamos a programar un código con C++, nos basamos en el pseudocódigo de libro de la clase, lo cual logramos lo siguiente.

```
void InsertionSort(float* array, int size)
{
    float temp;
    int j;
    for(int i=0;i<size;i++){
        temp = array[i];
        j = i-1;
        while(j>=0 && array[j]>temp){
            array[j+1]=array[j];
            j--;
        }
        array[j+1] = temp;
    }
}
```

Para entender a mayor precisión el código, diré que representa cada variable, el tiempo es el elemento que elegiré para compararlo con los demás, j es para acotar el recorrido que hay que hacer y solo se compare con los valores que hay a la izquierda. Y con eso en mente elegimos nuestro primer elemento en nuestro primer for y en el while lo vamos comparando y posicionando. Esto hay que hacerlo sucesivamente hasta que al final tengamos nuestro arreglo ordenado.

3. Selección:

Este algoritmo es uno de los más simples de los presentados, el cual va recorriendo el arreglo de entrada, el cual comparando una posición con los otros números, buscando así, el lugar más conveniente en donde estar, este proceso se repite una cantidad de veces equivalente al tamaño del arreglo y variando la posición seleccionada.

Ahora pongamos sobre la mesa el siguiente código, el cual para entenderlo, hay que ver que representa cada una de sus variables, small es la posición del dato más pequeño buscado, y temp lo ocupo como auxiliar para intercambiar los valores y no se pierdan.

```
void SelectionSort(float* array, int size_of_array)
{
    float temp = 0.0f;
    int small;
    for(int i=0;i<size_of_array;i++){
        small = i;
        for(int j=i;j<size_of_array;j++){
            if(array[j]<array[small]){
                small = j;
            }
        }
        temp = array[i];
        array[i] = array[small];
        array[small] = temp;
    }
}
```

Figura 3: Función SelectionSort

4. QuickSort:

Este es un método de ordenación más popular, el cual aprovecha la recursividad junto a la estrategia de divide-y-conquista.

Primero se elige un pivote al azar, y desde ese pivote, dividido la lista en dos, luego los voy comparando con el pivote, para así mover los elementos en los lugares más convenientes

```
218  /* Split for QuickSort */
219  int split_qs(int i, int j, float* array)
220  {
221      int p = rand() % (j - i) + i;
222      float tmp;
223      while(i < j)
224      {
225          while(i < p && array[i] <= array[p])
226          {
227              i++;
228          }
229          while(j > p && array[j] >= array[p])
230          {
231              j--;
232          }
233          tmp = array[i];
234          array[i] = array[j];
235          array[j] = tmp;
236
237          if(i == p)
238          {
239              p = j;
240          }else
241          {
242              if(j == p)
243              {
244                  p = i;
245              }
246          }
247      }
248
249      return p;
250  }
```

Figura 4.1: función split_qs

Ahora, el proceso lo divido de nuevo para crear sub-arreglos y ahí empezar a ordenar los elementos

```
252  /* QuickSort */
253  void QuickSort(float* array, int i, int j)
254  {
255      if(i < j)
256      {
257          int k = split_qs(i, j, array);
258          QuickSort(array, i, k);
259          QuickSort(array, k + 1, j);
260      }
261  }
```

Figura 4.2: función QuickSort

5. MergeSort:

Este Algoritmo aprovecha la recursividad y la estrategia de divide-y-conquista, el cual implementaremos en código de la siguiente manera.

```
157  void Merge(float* array, int p, int q, int r) {
158
159      int n1 = q - p + 1; //tamaño del subarray de la izquierda
160      int n2 = r - q; //tamaño del subarray de la derecha
161
162      float * Left = new float[n1];
163      float * Right = new float[n2];
164
165      //guardar valores del array en subarrays
166      for (int i = 0; i < n1; i++)
167          Left[i] = array[p + i];
168      for (int j = 0; j < n2; j++)
169          Right[j] = array[q + 1 + j];
170
171      int i, j, k;
172      i = 0; //indice del subarray izquierdo
173      j = 0; //indice del subarray derecho
174      k = p;
175
176      while (i < n1 && j < n2) {
177          if (Left[i] <= Right[j]) {
178              array[k] = Left[i];
179              i++;
180              }
181          else {
182              array[k] = Right[j];
183              j++;
184              }
185          k++;
186          }
187
188      while (i < n1) {
189          array[k] = Left[i];
190          i++;
191          k++;
192          }
193
194      while (j < n2) {
195          array[k] = Right[j];
196          j++;
197          k++;
198          }
199
200      delete[] Right;
201      delete[] Left;
202
203
204      }
```

Figura 5.1: Función Merge

El código utilizado en la figura 5.1 es la base del Mergesort. Este algoritmo funciona dividiendo el array de variables en dos. Luego, utilizando recursividad, se dividen a la mitad las partes divididas anteriormente hasta que solo queden variables individuales.

Después de lograr dividir el array, se vuelven a juntar uno por uno ordenando los números en el proceso.

```
void MergeSort(float* array, int start, int end) {
    if (start < end) {
        int m = start + (end - start) / 2; // en donde corto

        MergeSort(array, start, m); //el array de la izquierda
        MergeSort(array, m + 1, end); //el array de la derecha

        Merge(array, start, m, end);

        }
    }
```

Figura 5.2: MergeSort

En la figura 5.2 se puede observar la utilización de recursividad para lograr utilizar el Mergesort.

6. RadixSort:

7. Resultados Experimentales y Discusión:

Ahora para medir el tiempo de cuanto se demora cada algoritmo ocuparemos la librería time.h (nombres de las librerías), lo haremos 3 veces cada uno con distintas cantidades de datos y sacamos el promedio de lo ingresamos en una tabla.

Los Algoritmos corrieron en un computador con un procesador ryzen 5 4600 H, de RAM 8GB y una Tarjeta Gráfica gtx 1650 (decir especificaciones del computador ql)

Cantidad de Datos	Tiempo (milisegundos)			
	InsertSort	SelectSort	QuickSort	MergeSort
10000	115,193	136,998	2,199	2,335
20000	272,674	327,913	2,811	3,008
30000	599,489	739,178	4,362	4,634
40000	1060,16	1269,52	5,904	6,363
50000	1650,35	1912,91	7,561	8,036
60000	2372,94	2522,78	9,217	9,869
70000	3215,08	4108,23	10,7867	11,6853
80000	4201,86	5859,15	12,494	13,4567
90000	5334,77	6129,44	14,2033	15,3867
100000	6572,95	8367,36	15,8737	17,1913
200000	26251,7	36118,2	33,4707	36,181
300000	58312,9	758556	51,792	55,92
400000	106589	144793	71,642	77,939
500000	166816	218836	91,89	99,7787
600000	239335	322207	107,5917	117,142
700000	322889	442115	128,199	153,73
800000	426774	526423	150,6	181,49
900000	538878	704561	166,105	205,44
1000000	661507	888511	188,235	228,963

Cuadro 4: Tabla de los resultados experimentales

Al observar los datos de la tabla podemos destacar la diferencia de tiempo que hay entre los distintos algoritmos, los cuales los primeros en terminar es el QuickSort, el segundo es el MergeSort, el tercero es InsertSort y el último es el SelectionSort.

Por ejemplo si vamos a ordenar 1000000, el InsertSort se demoró un total de 661507 milisegundos llegando así en 3er lugar, en el caso SelectSort se tardó un total de 888511 milisegundos lo que se resume en último lugar, por el otro lado, en el QuickSort gastó un total de tiempo equivalente a 188,235 dejándole así en 1er lugar, y no hay que olvidarse del MergeSort el cual anduvo a la par con QuickSort pero llego 2do lugar. Ahora los tiempos del MergeSort podrían verse afectados ya que utilizamos el heap en él también, ya que pedía una mayor cantidad de recursos al correrlo, y si solo

ocupaba el stack colapsaba su memoria lo que terminaba con un desgarrador error llamado segmento core.

8. Conclusiones:

La diferencias de tiempo que se puede demorar entre un algoritmo u otro para un mismo fin juega un papel importante, ya que con grandes cantidades de datos, los tiempos de espera se “disparan”, lo que la información importante llega atrasada y las acciones que se pueden tomar con esos datos, puede que ya no sirvan.