

Diseño de Experimentos sobre Algoritmos de Ordenamiento

Planeación y Realización

Lo siguiente evidencia la fase de Planeación y Realización realizada para la experimentación sobre distintos algoritmos de ordenamiento:

Delimitación del objeto de estudio

A pesar de sonar cómo una tarea sencilla y natural para las personas, el problema del ordenamiento en la computación ha sido uno que atrae diversos estudios. Esto se debe a la necesidad de resolverlo de manera simple y efectiva. Diversas propuestas se han dado a lo largo de la historia de la computación, y una de las primeras autoras de algoritmos de ordenamiento fue Betty Holberton, quien trabajaba en la ENIAC, alrededor de 1951.

Una de las maneras para medir la eficacia de algoritmos de ordenamiento es la notación asintótica, la cual indica, en función del tamaño del conjunto de valores a ordenarse, el tiempo (máximo o mínimo) que este tarda en retornarlo ordenado bajo el criterio especificado. Esta notación indica el número de instrucciones que ejecuta el algoritmo para ordenar el conjunto de datos, pues el tiempo que la computadora tarde en realizar dichas instrucciones se suele tomar cómo un factor no controlado o de impacto no tan alto como lo es su cantidad.

Elección de Objeto de estudio y Variables de Respuesta

Con este experimento se desea conocer la medida en que diversos factores pueden afectar el rendimiento de un algoritmo de ordenamiento, tomando en cuenta factores cómo su notación asintótica, la arquitectura de la máquina en que este funcione, el tamaño del conjunto de valores a ordenarse, entre otros. Es así cómo se define la unidad experimental cómo **el tiempo de ejecución de este algoritmo sobre un conjunto de valores dado**, pues se apunta a encontrar y minimizar los factores que puedan incrementar este tiempo.

La variable de respuesta escogida para este experimento se define cómo el tiempo transcurrido desde que se pasa el conjunto de valores al algoritmo, hasta que este retorne el conjunto de valores ya ordenado. Cómo estos algoritmos serán implementados en C#, se utilizará la clase Stopwatch del Namespace System.Diagnostics, la cual, similar a un cronometro real, mide el tiempo transcurrido en milisegundos desde las invocaciones a sus métodos .Start() y .Stop().

Ejemplo:

```
using System.Diagnostics;
// ...

Stopwatch sw = new Stopwatch();

sw.Start();

// ...

sw.Stop();

Console.WriteLine("Elapsed={0}", sw.Elapsed);
```

Tomado de <https://stackoverflow.com/questions/969290/exact-time-measurement-for-performance-testing>

Determinación de Factores de Estudio y sus niveles

El rendimiento de un algoritmo de ordenamiento puede ser afectado por diversos factores. A continuación, se listan los diversos factores controlables, no controlables y de estudio que se tomaron en cuenta en este experimento.

Factores Controlables

- Algoritmo de Ordenamiento cómo tal.
- Tamaño del arreglo de entrada.
- Estado de los valores en el arreglo.
- Procesador del computador donde se ejecuta el algoritmo.
- Cantidad de procesos que se están ejecutando en el computador mientras se ejecuta el algoritmo.
- Lenguaje de programación en que se implemente el algoritmo.
- Técnica de compilación usada sobre el código del algoritmo.

Factores no Controlables

A continuación, se listan los factores que pueden afectar el rendimiento de un algoritmo, pero que no pueden ser controlados directamente, junto a la forma en que se reduce su efecto:

- Tamaño de memoria RAM del computador donde se ejecuta el algoritmo:
Si bien este factor puede alterar la capacidad de almacenamiento que puede utilizar el algoritmo, cambiar el tamaño implica desmontar por completo la PC, aparte de tener que utilizar partes a las que no se tiene acceso en el momento. Para evitar este efecto, se ejecutarán las pruebas en computadoras con la misma cantidad de memoria RAM (4 GB).
- Sistema Operativo del dispositivo donde se ejecuta el algoritmo.
Adquirir distintos sistemas operativos implica un proceso de instalación y desinstalación completa sólo para probar el rendimiento. Por esto, se establece que las pruebas serán ejecutadas por completo en computadoras con Windows 10
- Temperatura del dispositivo
Dispositivos más calientes trabajan a menor rendimiento. Se tratará de ejecutar estos algoritmos en una temperatura ambiente, sin que otros procesos que puedan elevar su temperatura estén ocurriendo al mismo tiempo.

Factores de Estudio

A continuación, se listan los factores de estudio, y sus niveles, para este experimento:

1. Algoritmo de Ordenamiento como tal:
Bien es sabido que no todos los algoritmos de ordenamiento funcionan de la misma manera. Es por esto por lo que se pretende utilizar dos algoritmos distintos, siempre que ambos tengan una complejidad temporal igual en notación asintótica.
2. Tamaño del conjunto de valores a ordenarse:
El principal factor que afecta un algoritmo. Es simple lógica, pues un tamaño de arreglo mayor implica que el algoritmo deberá ejecutarse más veces hasta que complete su tarea. Para este factor se escogió utilizar arreglos con tamaños de 10^1 , 10^3 y 10^5 entradas.
3. Estado de los valores en el arreglo a ordenarse:
Diversos algoritmos han probado tener mejores o peores rendimientos dependiendo del estado de sus entradas. Ejemplo de esto es Bubble Sort, cuya peor ejecución se da cuando los valores del arreglo ya están ordenados en el orden contrario al que se desea. Para medir la forma en que este factor afecta el rendimiento del algoritmo, se escogió que se le pondría a trabajar con datos ordenados ascendente y descendente, junto a datos completamente aleatorios.
4. Potencia del procesador que ejecute el algoritmo:
Procesadores con mayor potencia ejecutan las instrucciones del algoritmo de ordenamiento a mayor velocidad, por lo que se planea ejecutar el algoritmo en 3 procesadores distintos. Estos tendrán una potencia, según las especificaciones de sus respectivos fabricantes, de 1.9, 2.5 y 3,8 GHz.

Tratamientos de Experimentación:

Con los objetos de estudio definidos, estos serán los tratamientos que se usarán para el experimento:

Tratamiento	Algoritmo	Tamaño	Estado	Procesador (GHz)
1	Radix	10^1	Ascendente	1.6
2	Radix	10^1	Ascendente	2.5
3	Radix	10^1	Ascendente	3.2
4	Radix	10^1	Descendente	1.6
5	Radix	10^1	Descendente	2.5
6	Radix	10^1	Descendente	3.2
7	Radix	10^1	Aleatorio	1.6
8	Radix	10^1	Aleatorio	2.5
9	Radix	10^1	Aleatorio	3.2
10	Radix	10^3	Ascendente	1.6
11	Radix	10^3	Ascendente	2.5
12	Radix	10^3	Ascendente	3.2
13	Radix	10^3	Descendente	1.6
14	Radix	10^3	Descendente	2.5
15	Radix	10^3	Descendente	3.2

16	Radix	10^3	Aleatorio	1.6
17	Radix	10^3	Aleatorio	2.5
18	Radix	10^3	Aleatorio	3.2
19	Radix	10^5	Ascendente	1.6
20	Radix	10^5	Ascendente	2.5
21	Radix	10^5	Ascendente	3.2
22	Radix	10^5	Descendente	1.6
23	Radix	10^5	Descendente	2.5
24	Radix	10^5	Descendente	3.2
25	Radix	10^5	Aleatorio	1.6
26	Radix	10^5	Aleatorio	2.5
27	Radix	10^5	Aleatorio	3.2
27	Counting	10^1	Ascendente	1.6
28	Counting	10^1	Ascendente	2.5
29	Counting	10^1	Ascendente	3.2
30	Counting	10^1	Descendente	1.6
31	Counting	10^1	Descendente	2.5
32	Counting	10^1	Descendente	3.2
33	Counting	10^1	Aleatorio	1.6
34	Counting	10^1	Aleatorio	2.5
35	Counting	10^1	Aleatorio	3.8
36	Counting	10^3	Ascendente	1.6
37	Counting	10^3	Ascendente	2.5
38	Counting	10^3	Ascendente	3.2
39	Counting	10^3	Descendente	1.6
40	Counting	10^3	Descendente	2.5
41	Counting	10^3	Descendente	3.2
42	Counting	10^3	Aleatorio	1.6
43	Counting	10^3	Aleatorio	2.5
44	Counting	10^3	Aleatorio	3.2
45	Counting	10^5	Ascendente	1.6
46	Counting	10^5	Ascendente	2.5
47	Counting	10^5	Ascendente	3.2
48	Counting	10^5	Descendente	1.6
49	Counting	10^5	Descendente	2.5
50	Counting	10^5	Descendente	3.2
51	Counting	10^5	Aleatorio	1.6
52	Counting	10^5	Aleatorio	2.5

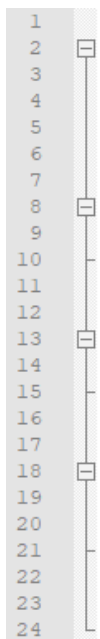
Planeación y Organización

Con lo planteado anteriormente, se procede a organizar un plan para obtener los datos de experimentación que se usarán:

1. Inicialmente se implementan los dos algoritmos, los cuales serán Radix Sort y Counting Sort. Cómo ambos deben ser realizados en C#, se utilizará el entorno de desarrollo Visual Studio 2019, pues este incluye per se herramientas que facilitan la medición del rendimiento en los programas que se desarrollen en él.
2. Una vez implementados los algoritmos, se procede a crear arreglos de diverso tamaño, y a llenarlos con números en órdenes ascendente, descendente y aleatorio.
3. Con los arreglos creados, se pasa a tomar una marca de tiempo, utilizando la clase Stopwatch, antes de iniciar el algoritmo.
4. Se ejecuta el algoritmo en uno de los tratamientos definidos.
5. Se toma otra marca de tiempo, una vez el algoritmo ha terminado su ejecución.
6. Ambas marcas de tiempo se comparan, utilizando el método .Elapsed de Stopwatch
7. El experimento se repite desde el paso 2, ahora con un tratamiento distinto. Continuando hasta que se completen los 53 tratamientos.

Análisis de la complejidad temporal y espacial de los algoritmos elegidos:

Count Sort:



The diagram on the left illustrates the steps of the Counting Sort algorithm. It shows a vertical timeline with numbered steps from 1 to 24. Key steps include:

- Step 1: Initialization of variables.
- Step 2: Finding the maximum value in the array.
- Step 3: Creating a count array of size max + 1.
- Step 4: Iterating through the input array to count occurrences of each element.
- Step 5: Calculating cumulative counts to determine the position of each element in the sorted array.
- Step 6: Placing each element into its correct position in the sorted array.
- Step 7: Returning the sorted array.

```

1 public static int[] countSort(int[] arr)
2 {
3
4     int[] count = new int[arr.Max() + 1];
5     int[] sorted = new int[arr.Length];
6
7     foreach (int x in arr)
8     {
9         count[x] += 1;
10    }
11
12    for (int i = 1; i < count.Length; i++)
13    {
14        count[i] += count[i - 1];
15    }
16
17    foreach (int x in arr)
18    {
19        sorted[count[x] - 1] = x;
20        count[x] -= 1;
21    }
22
23    return sorted;
24 }
  
```

N = arr.Length

$M = \text{arr.Max}() + 1$

Complejidad Temporal		Complejidad Espacial		
Línea	Repeticiones	Bytes	Repeticiones	Total
1	-	4	n	4n
4	1	4	m	4m
5	1	4	n	4n
7	n	4	1	4
9	n	-	-	-
12	m+1	4	1	4
14	m	-	-	-
17	n	4	1	4
19	n	-	-	-
20	n	-	-	-
23	1	-	-	-
Total:	5n + 2m + 4 -> O(n + m)	8n + 4m + 12 -> O(n + m)		

Radix Sort:

1		<code>public static void radixsort(int[] arr)</code>
2	☐	<code>{</code>
3		<code> int m = arr.Max();</code>
4		
5		<code> for (int exp = 1; m / exp > 0; exp *= 10)</code>
6		<code> countSortDigit(arr, exp);</code>
7		<code> }</code>
8		

$N = O(\text{countSortDigit})$

$M = \text{arr.Max}()$

Complejidad Temporal		Complejidad Espacial		
Línea	Repeticiones	Bytes	Repeticiones	Total
1	-	4	m	4m
3	1	4	1	4
5	m + 1	4	1	4
6	N * m	-	-	-
Total:	n * m -> O(n*m)	4m + 8 -> O(m)		

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

private static void countSortDigit(int[] arr, int exp)
{
    int[] output = new int[arr.Length];
    int[] count = new int[arr.Max()];

    for (int i = 0; i < arr.Length; i++)
        count[(arr[i] / exp) % 10]++;

    for (int i = 1; i < count.Length; i++)
        count[i] += count[i - 1];

    for (int i = (arr.Length) - 1; i >= 0; i--)
    {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (int i = 0; i < arr.Length; i++)
        arr[i] = output[i];
}

```

N = arr.Length

M = arr.Max()

Complejidad Temporal		Complejidad Espacial		
Línea	Repeticiones	Bytes	Repeticiones	Total
1	-	4	N + 1	4n + 4
3	1	4	n	4n
4	1	4	m	4m
6	N+1	4	1	4
7	n	-	-	-
9	M+1	4	1	4
10	m	-	-	-
12	N+1	4	1	4
14	n	-	-	-
15	n	-	-	-
17	N+1	4	1	4
18	n	-	-	-
Total:	7n + 2m + 6 -> O(n + m)		8n + 4m + 20 -> O(n + m)	

Realización del Experimento

Resultados de los tratamientos

1.6 Ghz	
Tratamiento	ticks
CountingSort 1.1	18151
CountingSort 1.2	67
CountingSort 1.3	12
CountingSort 2.1	470
CountingSort 2.2	434
CountingSort 2.3	382
CountingSort 3.1	26654
CountingSort 3.2	51865
CountingSort 3.3	33492
RadixSort 1.1	8854
RadixSort 1.2	66
RadixSort 1.3	64
RadixSort 2.1	4147
RadixSort 2.2	5769
RadixSort 2.3	4252
RadixSort 3.1	672109
RadixSort 3.2	622476
RadixSort 3.3	363866

2.5 Ghz	
Tratamiento	ticks
CountingSort 1.1	33435
CountingSort 1.2	44
CountingSort 1.3	17
CountingSort 2.1	386
CountingSort 2.2	1215
CountingSort 2.3	432
CountingSort 3.1	47382
CountingSort 3.2	82606
CountingSort 3.3	52701
RadixSort 1.1	7194
RadixSort 1.2	31
RadixSort 1.3	63
RadixSort 2.1	2923
RadixSort 2.2	2854
RadixSort 2.3	2682
RadixSort 3.1	639005
RadixSort 3.2	486915
RadixSort 3.3	452653

3.2 Ghz	
Tratamiento	ticks
CountingSort 1.1	7397
CountingSort 1.2	17
CountingSort 1.3	6
CountingSort 2.1	135
CountingSort 2.2	166
CountingSort 2.3	131
CountingSort 3.1	14806
CountingSort 3.2	28665
CountingSort 3.3	18710
RadixSort 1.1	2875
RadixSort 1.2	18
RadixSort 1.3	17
RadixSort 2.1	1224
RadixSort 2.2	1195
RadixSort 2.3	1189
RadixSort 3.1	188449
RadixSort 3.2	185110
RadixSort 3.3	198791

Analysis

Tabla de ticks promedio que le tomo a cada procesador para procesar los dos algoritmos elegidos:

Procesador	Counting	Radix	Counting/Radix
1.6 GHz	14614	186845	7.82%
2.5 GHz	24246	71999	33.68%
3.2 GHz	7781	64319	12.10%

Se puede observar que el algoritmo Counting tuvo un mejor rendimiento que el Radix por un margen muy considerable. En el procesador de 1.6 GHz, el algoritmo Radix tardó más de 10 veces lo que se tardó el Counting. En el de 2.5 GHz, el Radix tardó aproximadamente 3 veces lo que se demoró el Counting y, por último, en el procesador de 3.2 GHz, el algoritmo Counting tardó solo el 12.1% de lo que tardó el Radix.

Tabla de comparación del procesador de 3.2 GHz Vs 1.6 y 2.5 GHz.

Procesador	1.6 GHz	2.5 GHz
Promedio	36.72%	35.61%
	Promedio	36.17%

En esta tabla observamos que los procesadores de 1.6GHz y de 2.5GHz tienen un rendimiento similar, pues rinden a un porcentaje similar al compararlos con el de 3.2GHz. Sin embargo, el procesador de 3.2GHz tuvo un rendimiento notablemente superior, siendo casi 3 veces más rápido que los otros dos.

Control y Conclusiones Finales

Con base en las interpretaciones anteriores, concluimos que el mejor algoritmo de ordenamiento para los tratamientos elegidos es el Counting Sort. Además, que existe una relación entre la frecuencia del procesador y el tiempo de ejecución de los algoritmos a partir de una frecuencia de 2.5GHz.