Proyecto Integrador

Taller 4: Diseño de experimentos

Nicolás Martínez, Cristian Molina, Juan Manuel Castillo y Álvaro Escobar

url github: https://github.com/nicolasmr21/ExperimentalSorting

Diseño experimental como estudio de la eficiencia de ordenamiento de un conjunto de valores

Planeación y realización

1.Delimitación el problema u objeto de estudio:

Se llevará a cabo una experimentación sobre el ordenamiento de conjuntos de enteros teniendo en cuenta varios factores como el algoritmo usado, el tamaño de la entrada y el estado de sus valores. Todo esto, para saber cómo afectan la eficiencia medida en tiempo de tal proceso.

***Unidad experimental**: Conjuntos de enteros. Tales se pondrán a prueba para analizar el tiempo que se tardan en adquirir el carácter de orden.*

***2.Elección de la variable de respuesta que será medida en cada punto del diseño:** La variable de respuesta que se considera en este experimento es la eficiencia medida como el tiempo que tarda en realizarse el proceso de ordenamiento.*

3.Factores a estudiar de acuerdo a la supuesta influencia que tienen sobre la respuesta:

Factores controlables:

- *Sistema operativo.*
- *Lenguaje de programación.*
- *Algoritmo de ordenamiento.*
- RAM del computador donde se ejecuta el algoritmo.
- Estado de los valores en el arreglo.

Factores no controlables:

- *Estado del equipo en el que se realizarán las pruebas.*
- *Temperatura instantánea del equipo en el momento de prueba.*

Los factores de estudio a considerar son:

- Algoritmo de Ordenamiento (Radix sort o Counting sort).
- Tamaño del arreglo (10^1, 10^2, 10^3, 10^4, etc)
- Estado de los valores en el arreglo (en orden aleatorio, ordenado ascendente, ordenado descendente).

4.Niveles de cada factor, diseño experimental adecuado a los factores que se tienen y al objetivo del experimento:

Los diferentes valores que se asignan a cada factor estudiado en el diseño experimental son los siguientes

- Algoritmo de Ordenamiento contará con dos niveles: (1) Radix sort y (2) Counting sort.
- Tamaño del arreglo contará con dos niveles: (1)Entrada pequeña con máximo 10^5 elementos y (2)Entrada grande entre 10^5 minimo y 10^7 elementos maximo.
- Estado de los valores en el arreglo contará con tres niveles: (1) orden aleatorio, (2)ordenado ascendente y (3) ordenado descendente.

Nivel de Ordenamiento	***Nivel de tamaño***	***Nivel de estado de valores***	***TRATAMIENTO***
1	***1***	***1***	***1***
1	***1***	***2***	***2***
1	***1***	***3***	***3***
1	***2***	***1***	***4***
1	***2***	***2***	***5***
1	***2***	***3***	***6***
2	***1***	***1***	***7***
2	***1***	***2***	***8***
2	***1***	***3***	***9***
2	***2***	***1***	***10***
2	***2***	***2***	***11***
2	***2***	***3***	***12***

5.Planeacion y organizacion del trabajo experimental: con base en el diseño seleccionado se ha decidido utilizar el software ExperimentalSorting para hacer las pruebas de forma de que cada tratamiento se realice de forma adecuada y se realicen 12 repeticiones de cada uno, al final se realiza el registro del tiempo que tomo cada prueba y se consolida en un documento de excel. Por último, se hará uso de la herramienta spss para hacer el análisis de medias ANOVA de los tratamientos.

6.Realización del experimento:

En la siguiente tabla se presenta los tiempos en segundos obtenidos en cada uno de los tratamientos:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
R1	0,0329402	0,0219707	0,0179498	3,8772173	2,4445392	2,2917485	0,0149562	0,0049874	0,0029908	1,0646762	0,2283551	0,1994659
R2	0,0319366	0,018953	0,0189806	3,8727586	2,8970683	2,4336924	0,0089698	0,0039884	0,0009964	0,8331824	0,214937	0,2303846
R3	0,0329115	0,018915	0,0179223	3,9340418	2,8933496	2,4190329	0,0099749	0,002992	0,0019963	0,7385375	0,2154232	0,2642928
R4	0,0339124	0,0189776	0,0179517	3,9648045	2,612852	2,3055696	0,0070124	0,0029938	0,0019948	0,712746	0,2215821	0,5455475
R5	0,031877	0,0189217	0,0189478	3,9300919	2,8452804	2,3537192	0,0089423	0,001994	0,0019947	0,7631143	0,2564168	0,2159187
R6	0,0349052	0,017977	0,0189496	3,940369	2,2416154	2,4660288	0,0089766	0,0019948	0,0020242	0,7669674	0,2708054	0,2234038
R7	0,0319596	0,0199219	0,0179518	3,9032063	3,0321386	2,3141631	0,0079787	0,0019936	0,0019635	1,0422159	0,2543221	0,2094397
R8	0,0318713	0,0209426	0,0209452	3,8907246	2,5025741	2,5436957	0,0079805	0,0029916	0,0020276	0,8161349	0,2553159	0,2094704
R9	0,0349568	0,018981	0,0209425	3,9339497	2,4054248	2,5563455	0,0139603	0,0019962	0,0019932	0,7898889	0,2403557	0,2163918
R10	0,0368697	0,0229086	0,0179528	3,8991964	2,3382618	2,4290312	0,0079795	0,0019941	0,0019944	0,8148217	0,2363685	0,2822438
R11	0,0379367	0,0329126	0,0189486	3,8900658	2,3765009	2,3581792	0,0079459	0,0019955	0,0019967	0,7645442	0,208358	0,2179255
R12	0,0318765	0,0189486	0,0189496	3,8945714	2,3636024	2,3981616	0,0100036	0,001994	0,0009956	1,0252817	0,2159273	0,2134965

<u>Análisis</u>

Apoyándonos en el libro: Diseño Estadístico de Experimentos Ana María Lara Porras Profesora del Dpto. Estadística e IO de la Universidad de Granada (España) tenemos los siguientes supuestos:

Diseños factoriales con tres factores
Supongamos que hay a niveles para el factor A, b niveles del factor B y c niveles para el factor C y que cada réplica del experimento contiene todas las posibles combinaciones de tratamientos, es decir contiene los abc tratamientos posibles.

El modelo matemático que planteamos es el siguiente:

$$y_{ijk} = \mu + \tau_i + \beta_j + \gamma_k + (\tau\beta)_{ij} + (\tau\gamma)_{ik} + (\beta\gamma)_{jk} + u_{ijk}\ ,\ i = 1,2,3;\ j = 1,2;\ k = 1,2 \quad \text{, donde}$$

yijk: Representa la el tiempo de ordenamiento objetivo según el algoritmo i de ordenamiento, al tamaño de entrada j y al estado de los elementos k.
μ: Efecto constante, común a todos los niveles de los factores, denominado media global.
τi: Efecto medio producido por el algoritmo i de ordenamiento.
βj: Efecto medio producido por el tamaño de la entrada j.
γk: Efecto producido por el estado de los elementos k.
(τβ)ij: Efecto medio producido por la interacción entre el algoritmo i de ordenamiento. y el tamaño de la entrada j.
(τγ)ik: Efecto producido por la interacción entre el algoritmo i de ordenamiento y el estado de los elementos k.
(βγ)jk: Efecto producido por la interacción entre el tamaño de la entrada j y el estado de los elementos k..
(τβγ)ijk: Efecto producido por la interacción entre los tres factores.

La variable respuesta de este experimento tiempo de ordenamiento que se toma un determinado algoritmo, tamaño de entrada o estado en procesar un conjunto de enteros, siendo dicho conjunto la unidad experimental.

Se utiliza la prueba ANOVA en spss y se obtiene el siguiente resultado:

Pruebas de efectos inter-sujetos

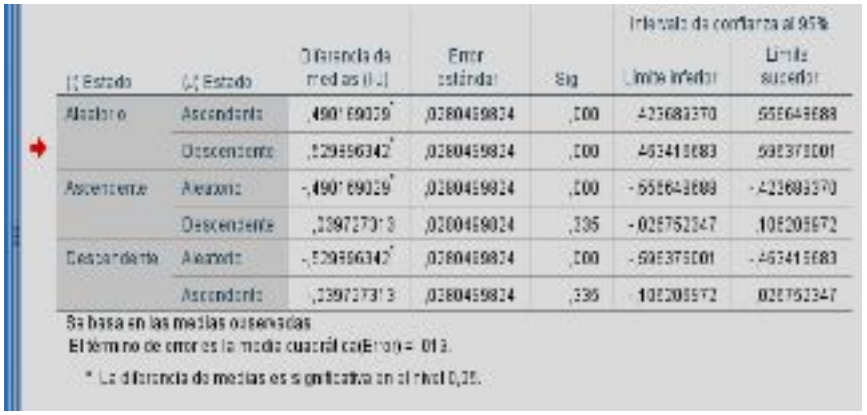
Variable dependiente: Tiempo

Origen	Tipo III de suma de cuadrados	gl	Media cuadrática	F	Sig.
Modelo corregido	235,071[a]	9	26,119	1303,185	,000
Intersección	106,403	1	106,403	5634,774	,000
Algoritmo	58,123	1	58,123	3078,039	,000
Entrada	102,822	1	102,822	5446,163	,000
Estado	8,362	2	4,181	221,417	,000
Algoritmo * Entrada	56,330	1	56,330	2983,085	,000
Algoritmo * Estado	1,414	2	,707	37,434	,000
Entrada * Estado	8,019	2	4,010	212,338	,000
Error	2,530	134	,019		
Total	344,004	144			
Total corregido	237,601	143			

a. R al cuadrado = ,989 (R al cuadrado ajustada = ,989)

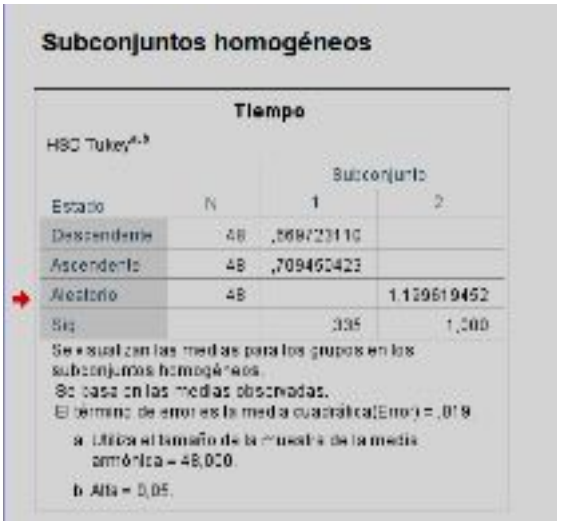
La Tabla ANOVA muestra las filas de Algoritmo, Entrada, Estado, Algoritmo*Entrada, Algoritmo*Estado y Entrada*Estado que corresponden a la variabilidad debida a los efectos de cada uno de los factores y a las interacciones de orden dos entre ambos.
En dicha Tabla se indica que para un nivel de significación del 5% todos los efectos son significativos para la variable de respuesta, por lo tanto es en este modelo donde vamos a realizar el estudio.

En primer lugar estudiamos qué estados de orden de los elementos son significativamente diferentes mediante el método de Tukey, se hace una prueba post hoc.



(I) Estado	(J) Estado	Diferencia de medias (I-J)	Error estándar	Sig.	Intervalo de confianza al 95% Límite inferior	Límite superior
Aleatorio	Ascendente	,490[*]	,0280459824	,000	,423683370	,556643688
	Descendente	,529356342[*]	,0280459824	,000	,463413083	,595299601
Ascendente	Aleatorio	-,490[*]	,0280459824	,000	-,556643688	-,423683370
	Descendente	,03972731 3	,0280459824	,335	-,028752347	,108206972
Descendente	Aleatorio	-,529356342[*]	,0280459824	,000	-,595299601	-,463413083
	Ascendente	,03972731 3	,0280459824	,335	-108206972	,028752347

Se basa en las medias observadas
El término de error es la media cuadrática(Error) = 013.
* La diferencia de medias es significativa en el nivel 0,05.



Subconjuntos homogéneos

Tiempo

HSD Tukey[a,b]

Estado	N	Subconjunto 1	Subconjunto 2
Descendente	48	,669723110	
Ascendente	48	,709460423	
Aleatorio	48		1,139619452
Sig.		,335	1,000

Se visualizan las medias para los grupos en los subconjuntos homogéneos.
Se basa en las medias observadas.
El término de error es la media cuadrática(Error) = ,019.

a. Utiliza el tamaño de la muestra de la media armónica = 48,000.

b. Alfa = 0,05.

Comprobamos que el estado de los elementos que produce más desviación en el tiempo de ordenamiento es el orden descendente y el que menos la produce es el ascendente. Por lo anterior, puede comprobar que un computador necesita más tiempo para ordenar un conjunto de números ordenados ascendentemente y sucede lo contrario con los que están ordenados descendentemente.

Los factores Algoritmo y Tamaño tienen cada uno dos niveles por lo tanto no se puede aplicar ningún método de comparaciones múltiples para comprobar qué tipo de Algoritmo de ordenamiento y qué Tamaño de entrada produce mayor/menor desviación en el llenado de las botellas. Podemos resolverlo calculando los llenados medios de cada uno de los niveles de los factores.

Descriptivos

	Entrada			Estadístico	Error estándar
Tiempo	Pequeño	Media		0[illegible]	,[illegible]
		95% de intervalo de confianza para la media	Límite inferior	0[illegible]	
			Límite superior	0[illegible]	
		Media recortada al 5%		0[illegible]	
		Mediana		0[illegible]	
		Varianza		,[illegible]	
		Desviación estándar		,[illegible]	
		Mínimo		[illegible]	
		Máximo		[illegible]	
		Rango		[illegible]	
		Rango intercuartil		[illegible]	
		Asimetría		,462	,283
		Curtosis		-,[illegible]	,559
	Grande	Media		1,[illegible]	,[illegible]
		95% de intervalo de confianza para la media	Límite inferior	1,380855803	
			Límite superior	2,028361761	
		Media recortada al 5%		1,[illegible]	
		Mediana		1,[illegible]	
		Varianza		1,898	
		Desviación estándar		1,37773977[illegible]	
		Mínimo		[illegible]	
		Máximo		[illegible]	
		Rango		3,7853387	
		Rango intercuartil		2,2986[illegible]25	
		Asimetría		,333	,283

Descriptivos de el efecto del tamaño de la entrada

Descriptivos

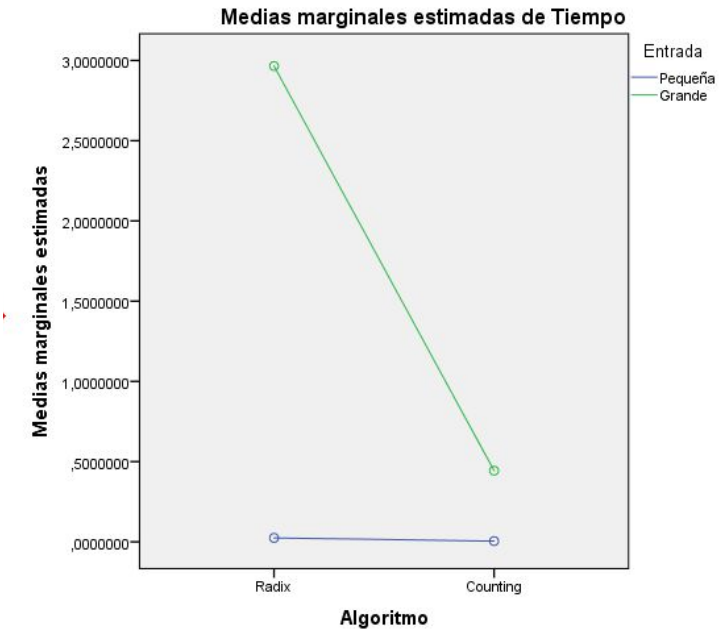
	Algoritmo			Estadístico	Error estándar
Tiempo	Radix	Media		1,494911858	,7835787830
		95% de intervalo de confianza para la media	Límite inferior	1,125282012	
			Límite superior	1,864541705	
		Media recortada al 5%		1,442895829	
		Mediana		1,136772850	
		Varianza		2,434	
		Desviación estándar		1,5602948948	
		Mínimo		,2175223	
		Máximo		5,3648848	
		Rango		5,3466823	
		Rango intercuartil		2,5332060	
		Asimetría		,913	,293
		Curtosis		-1,658	,559
	Counting	Media		,224275485	,1981340581
		95% de intervalo de confianza para la media	Límite inferior	,152423800	
			Límite superior	,295127171	
		Media recortada al 5%		,193064797	
		Mediana		,107215850	
		Varianza		,093	
		Desviación estándar		,3057867540	
		Mínimo		,2003956	
		Máximo		1,2648760	
		Rango		1,2638804	
		Rango intercuartil		,2577881	
		Asimetría		1,446	,293

Descriptivos de el efecto del algoritmo de ordenamiento

Teniendo en cuenta las anteriores figuras, se puede observar que la media de tiempo del algoritmo Counting sort es menor que el de el Radix, por lo tanto puede que sea más eficiente. Además, se puede identificar que las entradas grandes aumentan significativamente el tiempo de ordenamiento.

Por último se hace un cruce entre el algoritmo y la entrada en donde se puede observar:

Gráficos de perfil



Ya que no se cruzan en ningún momento las medias marginales de los dos factores se puede concluir que en este caso la interaccion algoritmo*entrada no hace variar mucho el

tiempo de ordenamiento. Pero si se puede revisar que el algoritmo counting es más eficiente para ambas entradas.

7. Diseño de pruebas

Diseño de pruebas para entradas grandes y pequeñas:

ExperimentTest

Scene1(): Un arreglo destinado al algoritmo de ordenamiento Radix Sort (Opción 1), con un tamaño de 98.500 elementos, es decir, inferior al tamaño máximo establecido para entradas pequeñas (Opción 1). Su estado de valor será de orden aleatorio(Opción 1).

Scene2(): Un arreglo destinado al algoritmo de ordenamiento Radix Sort (Opción 1), con un tamaño de 86.200 elementos, es decir, inferior al tamaño máximo establecido para entradas pequeñas (Opción 1). Su estado de valor será de orden ascendente(Opción 2).

Scene3(): Un arreglo destinado al algoritmo de ordenamiento Radix Sort (Opción 1), con un tamaño de 54.100 elementos, es decir, inferior al tamaño máximo establecido para entradas pequeñas (Opción 1). Su estado de valor será de orden descendente(Opción 3).

Scene4(): Un arreglo destinado al algoritmo de ordenamiento Radix Sort (Opción 1), con un tamaño de 16’550.600 elementos, es decir, superior al tamaño mínimo establecido para entradas grandes (Opción 2). Su estado de valor será de orden aleatorio(Opción 1).

Scene5(): Un arreglo destinado al algoritmo de ordenamiento Radix Sort (Opción 1), con un tamaño de 12’470.000 elementos, es decir, superior al tamaño mínimo establecido para entradas grandes (Opción 2). Su estado de valor será de orden ascendente(Opción 2).

Scene6(): Un arreglo destinado al algoritmo de ordenamiento Radix Sort (Opción 1), con un tamaño de 24’300.000 elementos, es decir, superior al tamaño mínimo establecido para entradas grandes (Opción 2). Su estado de valor será de orden descendente(Opción 3). así sucesivamente.

Scene7(): Un arreglo destinado al algoritmo de ordenamiento Counting Sort(Opción 2), con un tamaño de 14.900 elementos, es decir, inferior al tamaño máximo establecido para entradas pequeñas (Opción 1). Su estado de valor será de orden aleatorio(Opción 1).

Scene8(): Un arreglo destinado al algoritmo de ordenamiento Counting Sort(Opción 2), con un tamaño de 22.800 elementos, es decir, inferior al tamaño máximo establecido para entradas pequeñas (Opción 1). Su estado de valor será de orden ascendente(Opción 2).

Scene9(): Un arreglo destinado al algoritmo de ordenamiento Counting Sort(Opción 2), con un tamaño de 74.000 elementos, es decir, inferior al tamaño máximo establecido para entradas pequeñas (Opción 1). Su estado de valor será de orden descendente(Opción 3).

Scene10(): Un arreglo destinado al algoritmo de ordenamiento Counting Sort(Opción 2), con un tamaño de 8’100.000 elementos, es decir, superior al tamaño mínimo establecido para entradas grandes (Opción 2). Su estado de valor será de orden aleatorio(Opción 1).

Scene11(): Un arreglo destinado al algoritmo de ordenamiento Counting Sort(Opción 2), con un tamaño de 18’900.000 elementos, es decir, superior al tamaño mínimo establecido para entradas grandes (Opción 2). Su estado de valor será de orden ascendente(Opción 2).

Scene12(): Un arreglo destinado al algoritmo de ordenamiento Counting Sort(Opción 2), con un tamaño de 20’900.000 elementos, es decir, superior al tamaño mínimo establecido

para entradas grandes (Opción 2). Su estado de valor será de orden descendente(Opción 3).

Método	Escenario	Entrada	Salida
generateBigArrayRandomOrder()	Ninguna.	Ninguna.	Se prueba que se genere correctamente un arreglo con tamaño mayor o igual 10^8 elementos con orden aleatorio.
GenerateSmallArrayRamdomOrder()	*Ninguna.*	Ninguna.	Se prueba que se genere correctamente un arreglo con tamaño inferior o igual 10^5 elementos con orden aleatorio.
generateBigArrayAscendentOrder()	Ninguna.	Ninguna.	Se prueba que se genere correctamente un arreglo con tamaño mayor o igual 10^8 elementos con orden ascendente.
generateSmallArrayAscendentOrder()	Ninguna.	Ninguna.	Se prueba que se genere correctamente un arreglo con tamaño inferior o igual 10^5 elementos con orden ascendente.

generateBigArrayDescendentOrder()	Ninguna.	Ninguna.	Se prueba que se genere correctamente un arreglo con tamaño mayor o igual 10^8 elementos con orden descendiente.
generateSmallArrayDescendentOrder()	Ninguna.	Ninguna.	Se prueba que se genere correctamente un arreglo con tamaño inferior o igual 10^5 elementos con orden descendiente.

TreatmentTest

Método	Escenario	Entrada	Salida
RadixSort()	*Scene1()*	Ninguna.	Se prueba la eficiencia del algoritmo RadixSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior a 10^5 entregado aleatoriamente.

RadixSort()	*Scene2()*	Ninguna.	Se prueba la eficiencia del algoritmo RadixSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior a 10^5 entregado ascendentemente.
RadixSort()	*Scene3()*	Ninguna.	Se prueba la eficiencia del algoritmo RadixSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior a 10^5 entregado descendentemente.
RadixSort()	*Scene4()*	Ninguna.	Se prueba la eficiencia del algoritmo RadixSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior o igual a 10^7 entregado aleatoriamente.

RadixSort()	*Scene5()*	Ninguna.	Se prueba la eficiencia del algoritmo RadixSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior o igual a 10^7 entregado ascendentemente.
RadixSort()	*Scene6()*	Ninguna.	Se prueba la eficiencia del algoritmo RadixSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior o igual a 10^7 entregado descendentemente.
CountingSort()	*Scene7()*	Ninguna.	Se prueba la eficiencia del algoritmo CountingSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior a 10^5 entregado aleatoriamente.

CountingSort()	*Scene8()*	Ninguna.	Se prueba la eficiencia del algoritmo CountingSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior a 10^5 entregado ascendentemente
CountingSort()	*Scene9()*	Ninguna.	Se prueba la eficiencia del algoritmo CountingSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior a 10^5 entregado descendentemente.
CountingSort()	*Scene10()*	Ninguna.	Se prueba la eficiencia del algoritmo CountingSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior o igual a 10^7 entregado aleatoriamente.

CountingSort()	*Scene11()*	Ninguna.	Se prueba la eficiencia del algoritmo CountingSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior o igual a 10^7 entregado ascendentemente.
CountingSort()	*Scene12()*	Ninguna.	Se prueba la eficiencia del algoritmo CountingSort al encontrarse bajo un escenario en el que se medirá su potencial(En tiempo) para un arreglo inferior o igual a 10^7 entregado descendentemente.

Diagrama de Clases

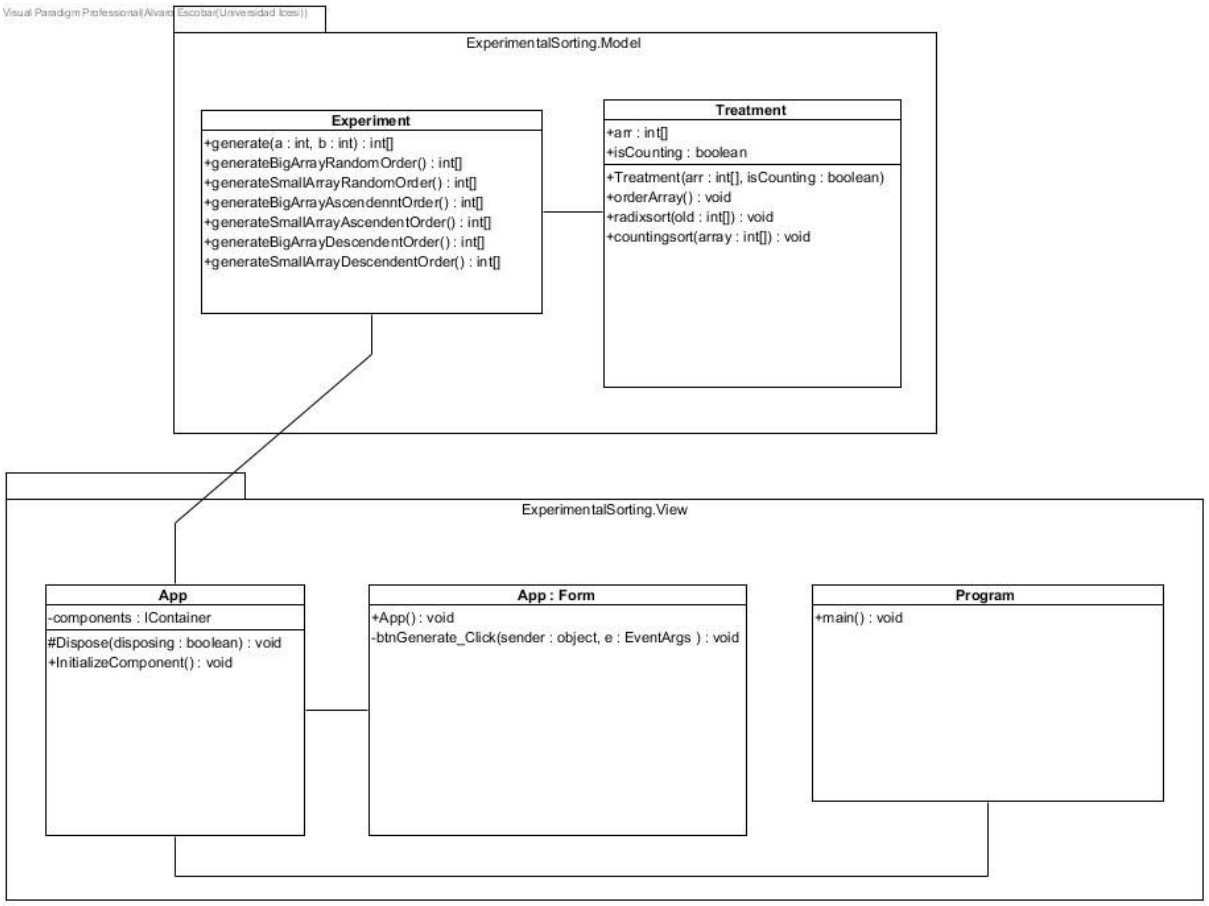
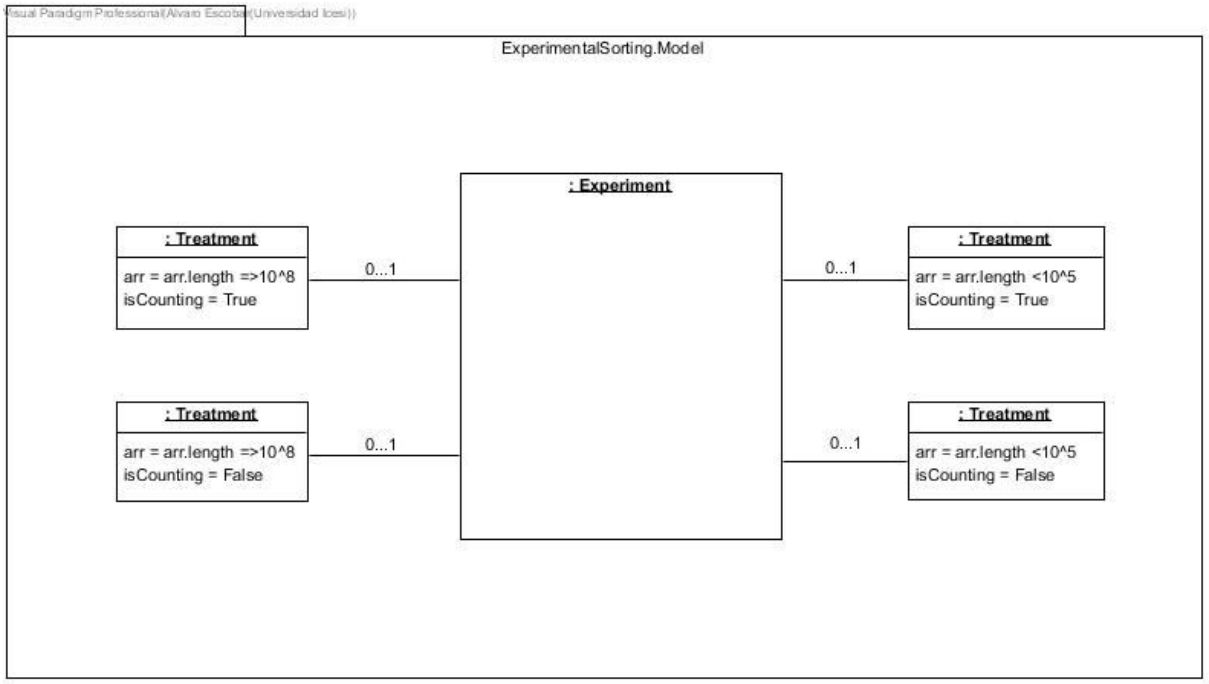


Diagrama de Objetos



Interpretación

Para obtener una visión más amplia del experimento que es llevado a cabo, es necesario dar una breve descripción de las variables las cuales componen este estableciendo una relación e interpretación con el enunciado propuesto.

Se asocia el concepto de **variable independiente** a aquellos factores en los cuales los integrantes encargados a resolver el enunciado, pueden dar uso para la variación de resultados de acuerdo a los experimentos (12 pruebas) implementados. Una de las variables independientes a considerar es el tamaño del arreglo ya que es adaptable y **controlable**, dependiendo de este, es posible obtener conclusiones distintas, es decir, un efecto sobre el experimento. Al ser independiente se requiere de un rango en el cual se pensó la construcción del código; Opción 1 para entradas pequeñas las cuales van entre 10^0 y 10^5, como también, Opción 2 para entradas grandes, establecidas entre 10^5 y 10^7.

En cuanto a **variables dependientes,** se tuvo en cuenta el procesador, RAM, e inclusive el entorno de trabajo, en este caso, Visual Studio. Como su nombre lo indica, son estas las dependientes del comportamiento de las independientes. Un ejemplo claro de esto es la relación establecida entre la memoria que es usada para la prueba unitaria de un método en la que si es excedida la memoria del ordenador debido a que el arreglo posee un tamaño demasiado grande, da paso a una excepción de “OutOfMemoryException”. Ya sea por la memoria destinada al programa de desarrollo Visual Studio o por la falta de memoria física que posee el equipo generarán errores que **no son controlables.**

Una vez explicadas las variables se da paso a las pruebas que brindarán los resultados estadísticos necesarios para las indagaciones y conclusiones oportunas en la búsqueda de solución del enunciado.

Es utilizada la **prueba ANOVA** para el análisis de la varianza, para este caso, no solo se dio interés por la varianza, sino también, para estudiar sus medias y la posibilidad de crear subconjuntos de grupos con medias iguales. Lo que conlleva a la diferenciación entre tiempos de cuál algoritmo podría obtener un mejor rendimiento dependiendo de los valores obtenidos en la experimentación.

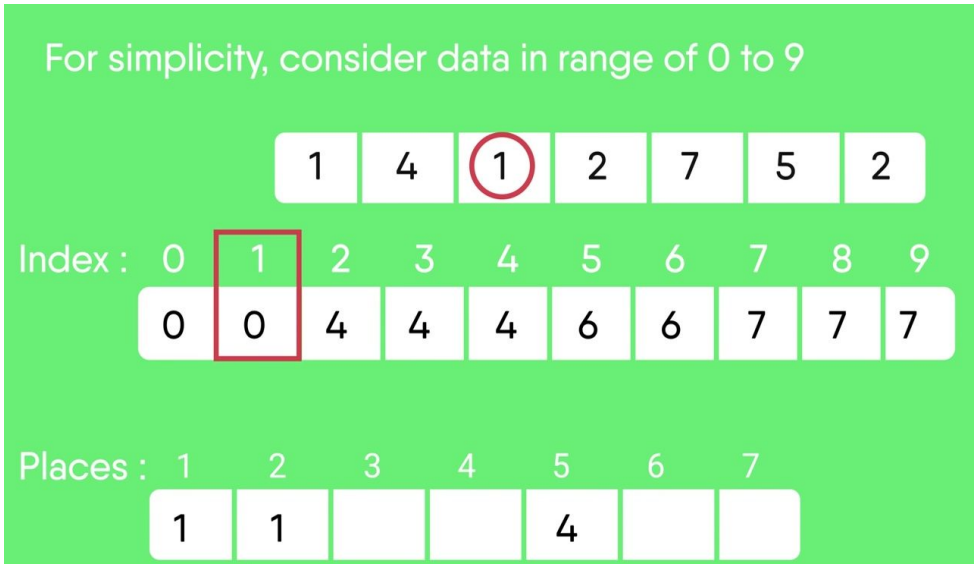
Posteriormente, visto que se ha determinado que se encontraron diferencias entre las medias, las pruebas de rango **post hoc** permiten determinar qué medias difieren. La prueba de rango post hoc identifica subconjuntos de medias que no se diferencian entre sí. Lo que otorga una visión más esclarecedora de donde se encuentran los datos que difieren en el experimento.

Respecto a los a los algoritmos utilizados en el desarrollo del taller se realizará una interpretación de estos. Cabe recalcar que son dos algoritmos permitidos por lo establecido en el enunciado evitando la repetición con otros grupos. Son algoritmos ya usados anteriormente en cursos pasados por lo que su implementación es más factible.

Counting Sort

Este es un algoritmo para ordenar solo elementos contables como los enteros en el cual se determina el número de elementos de la misma clase para luego ser ordenados. debe conocer el intervalo en el que estan los numeros para luego así crear un vector en el cual se van a introducir los números de cada clase. Es de propósito específico.

- Si se clasifican elementos no primitivos (objeto), se necesita otra matriz auxiliar para almacenar los elementos ordenados
- La clasificación de conteo se puede usar solo para ordenar valores discretos (por ejemplo, enteros), porque de lo contrario no se puede construir la matriz de frecuencias.

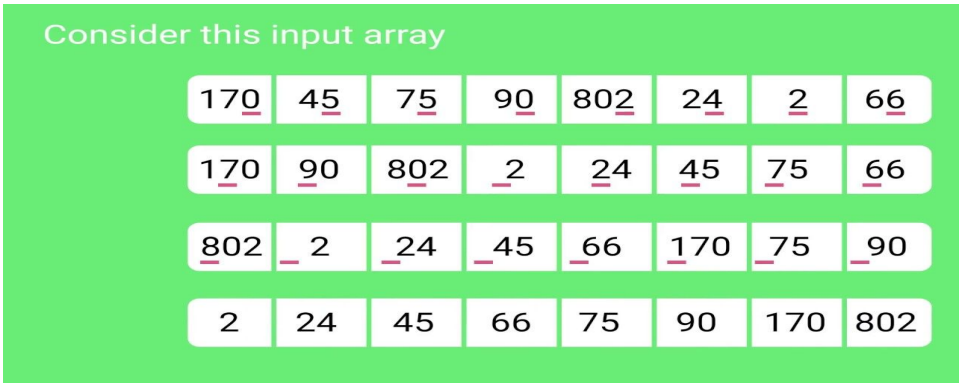


Fuente: ***https://www.geeksforgeeks.org/counting-sort/***

Radix

El límite inferior para el algoritmo de ordenación basado en la comparación es O (n * log n), de igual forma al de Quick Sort, Merge Sort y Heap Sort. Por lo tanto, el algoritmo es eficiente hasta que el número de dígitos (clave) sea menor que log n. Una gran diferencia entre Radix y Counting Sort es que en el segundo no se puede usar si un rango de valores clave es grande (suponiendo que el rango es de 1 a n2), por lo que Radix es la mejor opción para clasificar en tiempo lineal

- Radix Sort es estable, preservando la orden de elementos iguales.
- El tiempo de ordenar cada elemento es constante, ya que no se hacen comparaciones entre elementos.
- Radix Sort no brinda un funcionamiento del todo correcto cuando los números son muy largos, ya que el total de tiempo es proporcional a la longitud del número más grande y al número de elementos a ordenar.



Fuente: https://www.youtube.com/watch?v=nu4gDuFabIM/

Control y conclusiones finales

Teniendo en cuenta la información obtenida en este estudio gracias al análisis estadístico realizado anteriormente, se puede concluir lo siguiente sobre los factores de estudio que se definieron inicialmente.

1. **Algoritmo de Ordenamiento (Radix sort o Counting sort):** En la etapa de análisis se obtuvo como resultado que el algoritmo Counting Sort presentaba una media en el tiempo de ejecución menor en comparación a la del algoritmo Radix Sort, lo cual nos permite concluir que el Counting Sort es un poco más rápido, esto puede deberse a que la estrategia de ordenamiento utilizada por ambos algoritmos es diferente.

2. **Tamaño del arreglo (10^1, 10^2, 10^3, 10^4, etc):** El gráfico que presenta la relación entre las medias marginales de los tamaños de la entrada dependiendo del algoritmo de ordenamiento nos permite evidenciar que el tamaño de la entrada aunque es decisivo para determinar el tiempo de ejecución de cada algoritmo, no genera ningún cambio relevante en promedio debido a que ambos tardan un tiempo parecido que está dado por la cota $O(n+k)$.

3. **Estado de los valores en el arreglo:** La prueba ANOVA nos permite comparar la variabilidad del tiempo dependiendo del orden de los datos (Ascendente, Descendente y Aleatorio) y se obtuvo una desviación mayor para los datos de orden ascendente, por lo que dependiendo del orden natural de los datos el tiempo podrá aumentar o disminuir.

En conclusión, los factores de estudio que se definieron al inicio afectan de alguna forma el tiempo de ejecución del proceso, sin embargo, debido a que cuando se estudia la complejidad temporal de los algoritmos se toma como caso base el peor de los casos es decir la notación O , que nos da una cota más alta con respecto al tiempo exacto y para la cual un pequeño margen de error no resulta relevante, estos factores que aumentan tan ligeramente el tiempo de ejecución resultan despreciables.

Análisis de complejidad de los algoritmos:

Counting Sort:

Complejidad temporal:

	Constante	Mejor caso	Peor caso
private void countingsort(int[] array) {		Temporal	
int[] aux = new int[array.length];	C1	1	
int min = array[0];	C2	1	
int max = array[0];	C3	1	
for (int i = 1; i < array.length; i++) {	C4	n	
if (array[i] < min) {	C5	n-1	
min = array[i];	C6	1	(n-1)/2
} else if (array[i] > max) {	C7	n-1	
max = array[i];	C8	1	(n-1)/2
}			
}			
int[] counts = new int[max - min + 1];	C9	1	
for (int i = 0; i < array.length; i++) {	C10	n+1	
counts[array[i] - min]++;	C11	n	
}			
counts[0]--;	C12	1	
for (int i = 1; i < counts.length; i++) {	C13	k	
counts[i] = counts[i] + counts[i-1];	C14	k-1	
}			
for (int i = array.length - 1; i >= 0; i--) {	C15	n+1	
aux[counts[array[i] - min]--] = array[i];	C16	n	
}			
this.array = aux;	C17	1	
}			
		Total	8n+5+k
	Complejidad	O(n)	

Complejidad espacial:

Tipo	Nombre	Costo	Cantidad
Entrada	array	K1	n
Auxiliar	min	K2	1
	max	K2	1
	counts	K1	m
Salida	aux	K1	n
Complejidad O(n)			

Radix Sort:

Complejidad temporal:

	Constante	Mejor caso	Peor caso
public void radixsort(int[] old)		Repeticiones	
{			
int i, j;	C1	1	
int[] tmp = new int[old.Length];	C2	1	
for (int shift = 31; shift > -1; --shift)	C3	32	
{			
j = 0;	C4	31	
for (i = 0; i < old.Length; ++i)	C5	30n +1	
{			
bool move = (old[i] << shift) >= 0;	C6	30n	
if (shift == 0 ? !move : move)	C7	30n	
old[i - j] = old[i];	C8	30n	0
else	C9	0	30n
tmp[j++] = old[i];	C10	0	30n
}			
Array.Copy(tmp, 0, arr, arr.Length - j, j);	C11	31	
}			
}			
	Total:	97+120n	97+150n
Complejidad: O(n)			

Complejidad espacial:

Tipo	Nombre	Costo	Cantidad
Entrada	old	K1	n
Auxiliar	i	K2	1
	j	K3	1
	tmp	K4	1
	shift	K5	1
	move	K6	1
Salida	arr	K7	n
		Total	2n+5
Complejidad: O(n)			