Estructuras de Datos

Vectores vs Arrays, Analisis de Algunas de sus Operaciones

Héctor F. Jiménez S.
hfjimenez@utp.edu.co
PGP KEY ID: 0xB05AD7B8

Cristian Camilo Perilla C.
camilo980818@gmail.com

Fecha de Entrega: Septiembre, 2016
Profesor: Gustavo Adolfo Gutierrez Sabogal

1 Objetivos

- Identificar las ventajas y desventajas de los arrays y vectores.
- Identificar la complejidad de las operaciones realizadas sobre los arrays y vectores,como **add(), get(), remove(), insert()**.
- Realizar la implementacion y analisis en lenguaje C++11.

En nuestro mundo de algoritmos vivimos y tratamos de subsistir para poder dar soluciones a una gran cantidad de problemas del mundo real, somos nosotros los que con ideas poderosas y abstractas las podemos convertir en magia para tratar esa gran cantidad de información que esta allí, sin importar el tipo de patrón, dato o complejidad es necesario darle el respeto adecuado a los datos, por ello las estructuras de datos existen para brindarles el mejor cariño y manipulación que nos permitan realizar un tratamiento correcto, eficiente y preciso en la medida posible.
En este informe se plasman algunos de los resultados que fueron observados en la implementación de dos tipos de estructuras, específicamente de los muy famosos *arrays y vectores* ambos son estructuras de datos que permiten manipular datos en forma lineal dado que los elementos se encuentran en una secuencia definidos por una función lineal que relaciona los elementos de estos. Exploraremos algunas de las operaciones que se realizan con ellas, y su eficiencia temporal.

2 Vectores vs Array

Configuración para el Laboratorio

El setup utilizado para realizar este laboratorio ha sido creado en un entorno virtualizado que posee las siguientes características[1] :

- ***Sistema Operativo***: Debian Gnu/Linux,Wheezy.
- ***Cantidad de Memoria Ram***:2GB
- ***Procesador Usado***:Intel(R) Atom(TM) CPU N550 v3 @ 1.50GHz
- ***VCores Disponibles***:2
- ***Versión Compilador***:gcc version 6.1.1
- ***Arquitectura***:i686

2.1 Operación de Inserción

La operación de inserción en su forma generalizada **add(*i*)** se encarga de adicionar un elemento k en la posición i. Esta operación puede ser realizada en los arrays y vectores. asignaremos a todos las posiciones del vector de tamaño n(*aleatorio*) un valor constante(1).

1. La implementación de este código puede ser realizada en cualquier arquitectura pues la complejidad que mas nos interesa es la temporal que me indica cuanto se demora un algoritmo en terminar.

2.1.1 VectorInsertion vs ArrayInsertion

Las implementaciones de inserción fueron provistas por el profesor a cargo, para ser utilizadas en nuestra prueba de laboratorio:

```
void arrayInsertion(int n) {
  int *a = new int[n];
  for (int i = 0; i < n; i++) {
    a[i] = 1;
  }t
}
void vectorInsertion(int n) {
  Vector<int> a;
  for (int i = 0; i < n; i++) {
    a.add(i, 1);
  }
}
```

Código 1: Inserciones, Vector, Array.

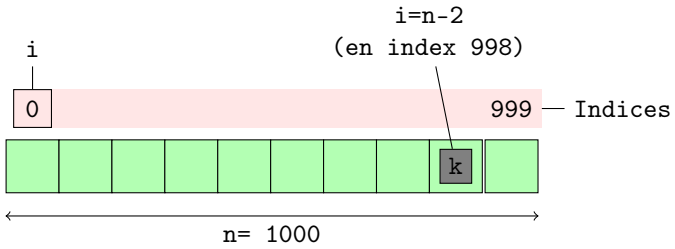
Para comparar ambas estructuras de datos,varias pruebas fueron realizadas dándonos un aporte significativo para nuestro análisis. Nosotros vector que tiene un tamaño n **100.** posiciones el cual fue generado tres veces de forma experimental.Dentro de cada posición $i-esima$ habrá un valor aleatorio $k-esimo$ obtenido de utilizar un generador pseudo aleatorio que se encuentra basado en el algoritmo de Mersenne Twister[2] que reemplaza la vieja función **rand()** de lenguaje C, para producir valores sin signo de 32[3],64 bits. La función implementada para generar los números pseudo aleatorios llamado **produce** recibe tres parámetros de tipo entere ellas la cantidad de valores aleatorios a generar**s**, el rango minimo y maximos dentro de los cuales se generara el valor pseudo aleatorio, la implementacion es la presentada en el Código 2.

2. https://en.wikipedia.org/wiki/Mersenne_Twister
3. cppreference,http://www.cplusplus.com/reference/random/mersenne_twister_engine/

```
Vector<int> produce(int s, int l, int u) {
  random_device rd;
  mt19937 gen(rd());
  uniform_int_distribution<> dis(l, u);
  Vector<int> result;
  for (int n = 0; n < s; n++) {
    result.add(n, dis(gen));
  }
  return result;
}
```

.

Código 2: Números Pseudo-aleatorios usando Merssene Twister



El procedimiento para obtener los datos experimentales fue llevada a cabo realizando el siguiente procedimiento:

1. producir el vector que tiene tamaño $n(100)$ con k valores aleatorios.
2. recorrer cada posición del vector anterior para obtener el valor k que hay en la posición i.
3. iterar 100 veces llamando a las funciones intermedias(*InsertionVector(k)*, *InsertionArray(k)*) que toma como argumento el valor k.
   (a) iniciar el cronometro
   (b) realizar la inserción de los k valores.
   (c) finalizar cronometro.
4. Obtener la media y la desviación estándar.(ns).

Los datos se encuentra alojados en github[4]. Para analizar los datos obtenidos fue necesario utilizar una hoja de calculo, los datos se encontraban en formato plano de la forma como se puede ver en la tabla 2.1 en desorden, de manera que para poder graficar los datos, ordenamos los datos de forma ascendente los

4. https://github.com/heticor915/DataStructures/tree/master/HomeWorks/Homework2-1/Solution/Data/ArrayvsVector

valores *kesimos* de los vectores y arrays. Nosotros graficaremos los **valores aleatorios** vs **tiempo de inserción**, hallados [5]

k	ArrayInsertion[ns]	VectorInsertion[ns]
k1	Tiempo$k1$ArrayInsertion	Tiempo$k1$VectorInsertion

Table 2.1: Formato de Datos

Después de esto lo que realizamos fue obtener las gráficas correspondientes y al notar que los valores k eran aleatorios realizamos un diagrama de dispersión como se puede ver en la figura 1.

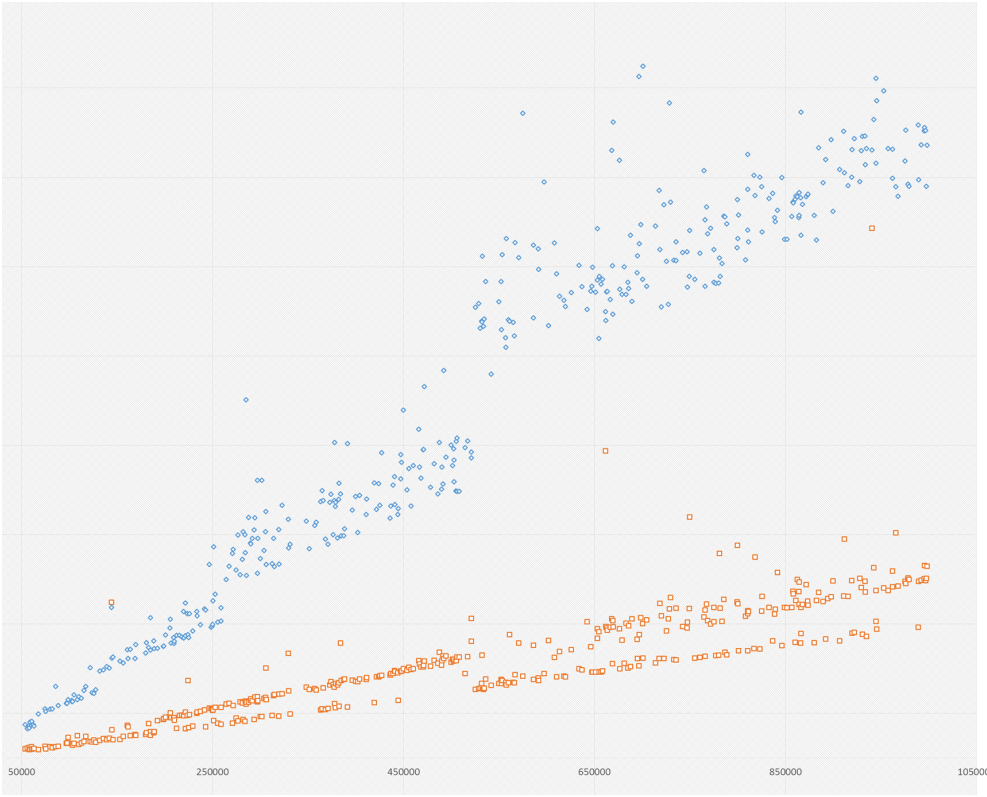


Figure 1: Diagrama de dispersión de valores aleatorios , tiempo de inserción.

Aunque un diagrama de dispersión permite formarse una primera impresión muy rápida sobre el tipo de relación existente entre las dos variables, utilizarlo como una forma de cuantificar esa relación tiene un serio problema para nosotros pues

5. `https://github.com/heticor915/DataStructures/tree/master/HomeWorks/Homework2-1/Solution/Data/ArrayvsVector`

en una situación ideal si tomáramos los puntos de la dispersión y los uniéramos debería darnos una linea recta, no tendríamos que preocuparnos de encontrar la recta que mejor se ajuste, pero en una nube de puntos dispersa como la presentada en la figura 1debemos realizar un ajuste. Realizar una regresión lineal para tener una mejor representación y modelo mas ajustado para el modelo es lo adecuado, además así se realiza en los laboratorios de física donde se busca un modelo mas acertado.
Nuestra hoja de calculo posee estas herramientas ya incluidas por lo que haremos uso de ellas, que nos entrega la ecuación de la mejor linea recta como se puede ver en la figura 2.

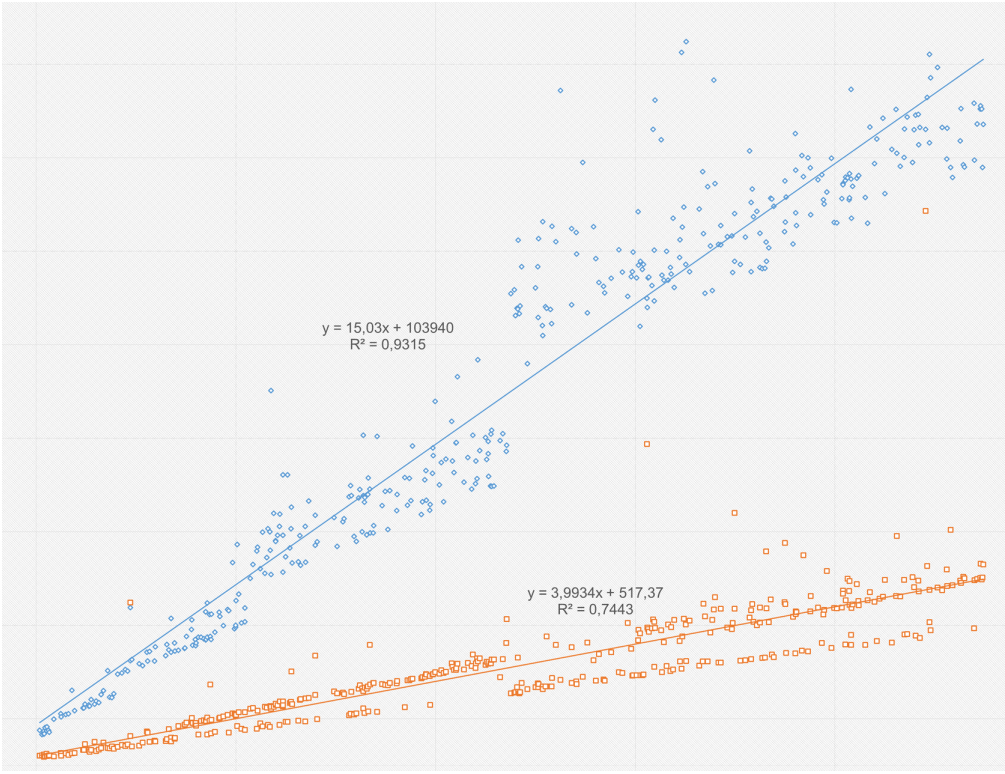


Figure 2: Regresión lineal.

El R^2 indica cuanto porcentaje la Ecuación de la linea recta se aproxima en la relación de las variables, esto es un parámetro importante que nosotros hemos utilizado para aceptar o rechazar la ecuación que describe la relación de los datos.Nosotros hemos aceptado la representación

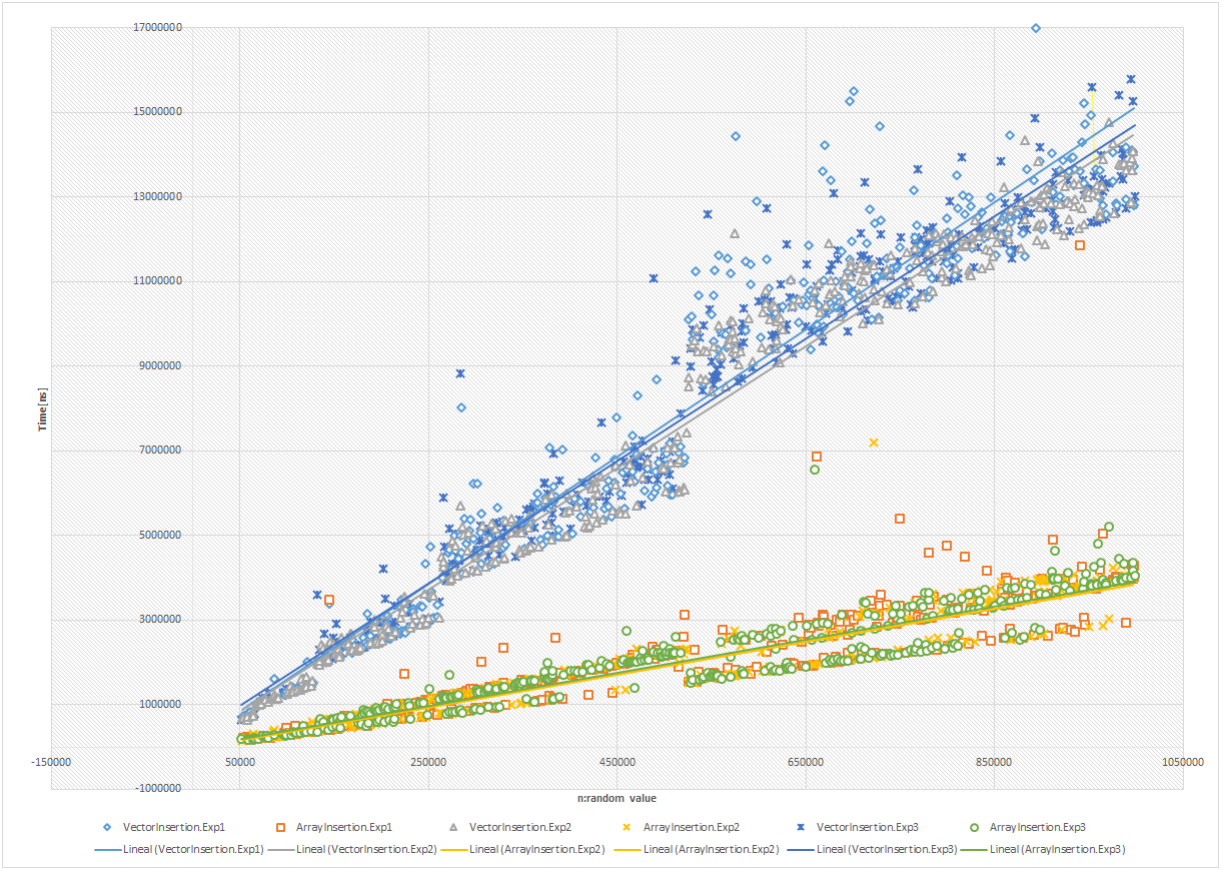


Figure 3: Gráfica de la Implementacion de Inserción para Vector y Array.

Para comprobar que tan rápidos son los arrays y los vectores nosotros graficamos los datos que se ven en la figura 3 que presenta la comparación de 3 experimentos con un vector de 400 posiciones rellenos de elementos aleatorios que describen el comportamiento de una distribución normal (o de Gauss) entre los rangos $l = 50000$ y $u = 1000000$, de los tres experimentos tenemos las siguientes valores promedio, y las siguientes desviaciones estandar:

	Promedio de Valores Aleatorios	***Desviacion Estandar Valores Aleatorios***
Experimento 1	516974,105	271697, 05
Experimento 2	513338,9425	287655,06
Experimento 3	523706,3025	275309,08

Table 2.2: Promedios, y desviaciones estándar de los valores aleatorios del experimento de la figura 3. ns

Donde se puede observar que el valor promedio para el experimento1,experimento2,experimento3 de valores aleatorios es 518006 de 400 datos, con una desviación promedio de +-278220.4[ns]. El comportamiento linea salta a la vista de las gráficas pues era una de las cosas esperadas en el comportamiento.
De la figura 3 se puede concluir que para los tres diferentes experimentos que se encuentran graficados en el mismo plano cartesiano, el comportamiento de sus homólogos tiene una tendencia muy parecida, en cuanto a la operación de inserción, los arrays son mucho mas eficientes respecto al tiempo, pues se puede observar que cuando el valor aleatorio a insertar es mayor en los arrays su tiempo es mucho menor comparado con la de los vectores.
Los arrays son estructuras de datos que no pueden expandir su tamaño de almacenamiento de forma dinámica por lo que tiene un tamaño fijo y nosotros como programadores debemos asegurarnos en resevar la memoria correspondiente a la cantidad de elementos que utilizaremos. insertar elementos en un array tiene una complejidad ($\mathcal{O}(n)$) con respecto al tamaño del array que estamos llenando, asumiendo que el array para llenarlo inicia en la posición **0**, es decir la operación de add para este punto siempre se va a realizar al final de la secuencia. Los vectores puede crecer dinamicamente, pero se debe realizar una operación de **resize** para nuestra prueba y solución de la pregunta 1.1 utilizamos la política por default de **ns=capacity*2**. Los datos según las figuras 3,

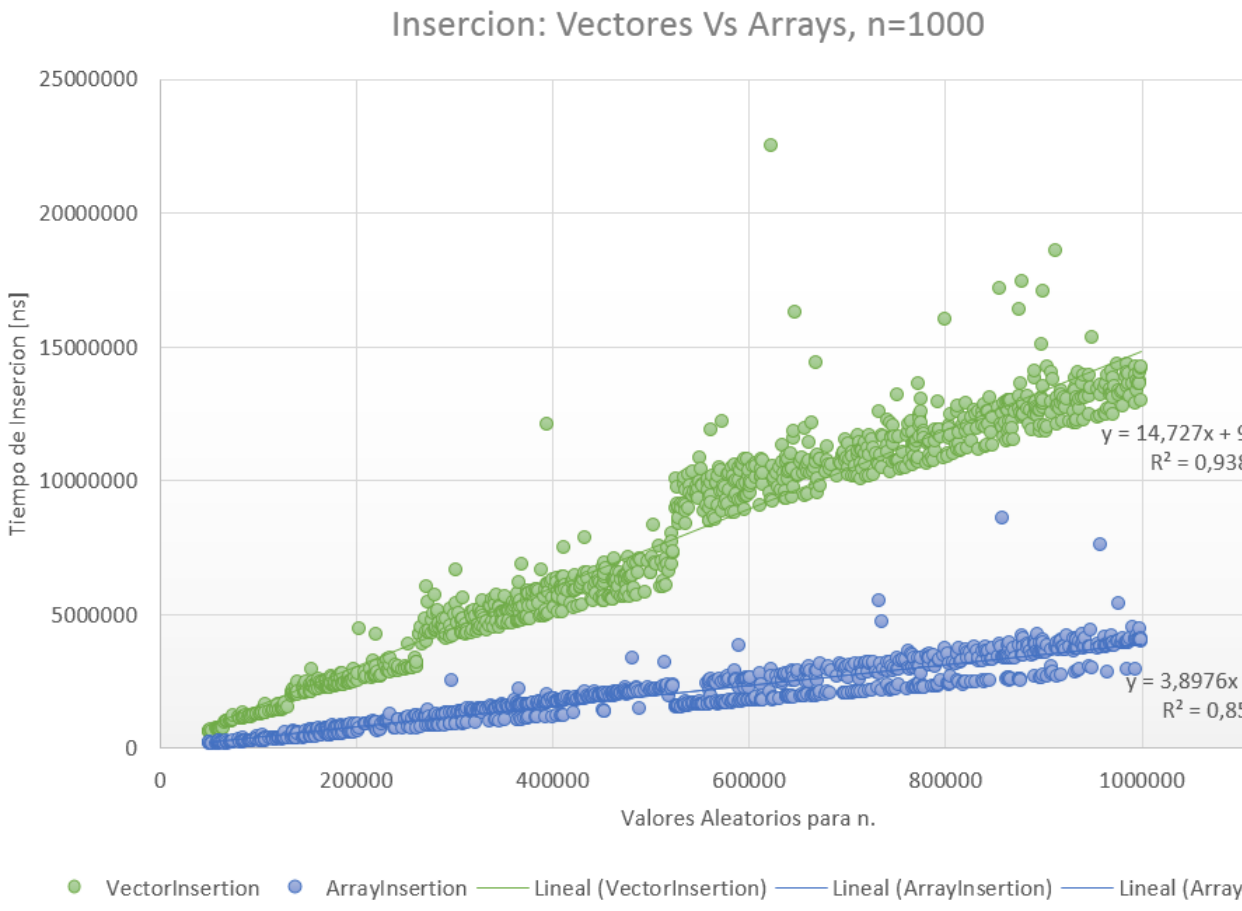


Figure 4: Experimento 2, n=1000

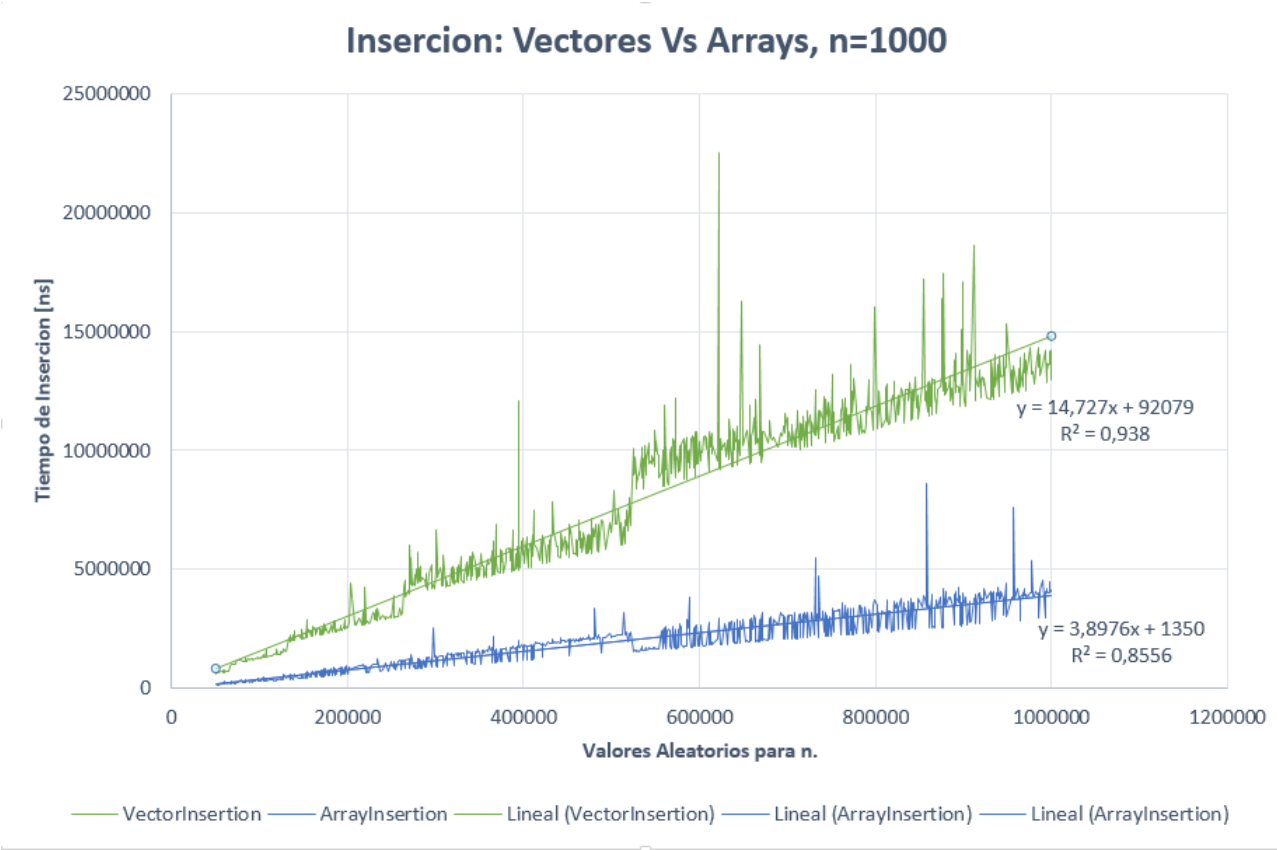


Figure 5: Diferencias, Experimento 1, n=1000

Para nuestro segundo experimento decidimos incrementar el tamaño del vector que contendría los valores aleatorios por un **n=1000** *véase figura 4,5*, Al tener mas datos nosotros como analistas y teniendo presente la teoría de los grandes números[6] podremos confiar mas en el comportamiento de los datos, como esto se trata de lidiar con datos y probabilidades, esta ley establece que al observar continuadamente en el tiempo un experimento cuyo resultado es aleatorio los resultados tienden a estabilizarse. Podemos afirmar de nuestras practicas experimentales la implementacion de inserción mas rápida en tiempo es la de array para esta parte la adicion de un elemento 1 es al final de la secuencia.

2.1.2 VectorInsertion con Diferentes Políticas de Crecimiento

Como sabemos nuestra clase vector tiene una gran diferencia con respecto a la implementacion de los arrays, vector es dinámico en memoria es decir que nosotros como programadores usando nuestra clase no necesitamos preocu-

6. `https://en.wikipedia.org/wiki/Law_of_large_numbers`

parnos por conocer la dimensión o tamaño finito de nuestra secuencia si no que el puede ir creciendo de manera lineal y en medida que nosotros necesitemos. Para este punto sabemos que agregaremos elementos siempre al final de la secuencia, pero cada vez que nosotros agregamos un elemento al vector la capacidad para almacenar elementos a ella se reducirá en una posición. Vector tendrá que ser inteligente es decir deberá utilizar una política de crecimiento para incrementar la reserva de espacio en memoria.
Considere la siguiente implementación, en la cual se puede observar el caso claramente del resize:

```
void add(int i, const T &x) {// recibimos la posición i y un valor x
        //para adicionar al vector.
  assert(i >= 0 && i <= sz);//si la posicion es un valor negativo
  //e i esta entre el tamaño del vector[1]
  if (sz == capacity) {
  //si ya tenemos el vector lleno y necesitamos agregar un elemento mas
    resize(); //redimensionamos el vector.
  }
  moveOneRight(i);        //corremos el valor i una posición a la derecha.
  storage[i] = x;               //ingresamos elemento x al vector.
  sz = sz + 1;                 //incremente el tamaño en 1
}
```

.

Código 3: Implementacion de Add

El resize se encargara de re dimensionar el vector con la política que tenga en el momento, luego deberá hacer una copia de seguridad para que los datos que habían previamente en el vector no se pierdan, y crear un vector nuevo con una capacidad nueva que pueda almacenar lo que necesita, luego una copia al nuevo vector y finalmente destruir el espacio de memoria en reserva que se uso como temporal para la copia. En esta subseccion lo que haremos sera jugar con estas políticas, los datos se generaran de la misma forma que se hizo previamente pero variando la política de crecimiento que utiliza **add**.
Las políticas que hemos puesto a prueba en nuestro laboratorio son :

- int ns = capacity * 1.5;
- int ns = capacity * 1.8;
- int ns = capacity * 3;
- int ns = capacity * 2;
- int ns = capacity + 1;

Realizando las gráficas de los datos obtenidos entre las diferentes políticas obtuvimos el siguiente gráfico:

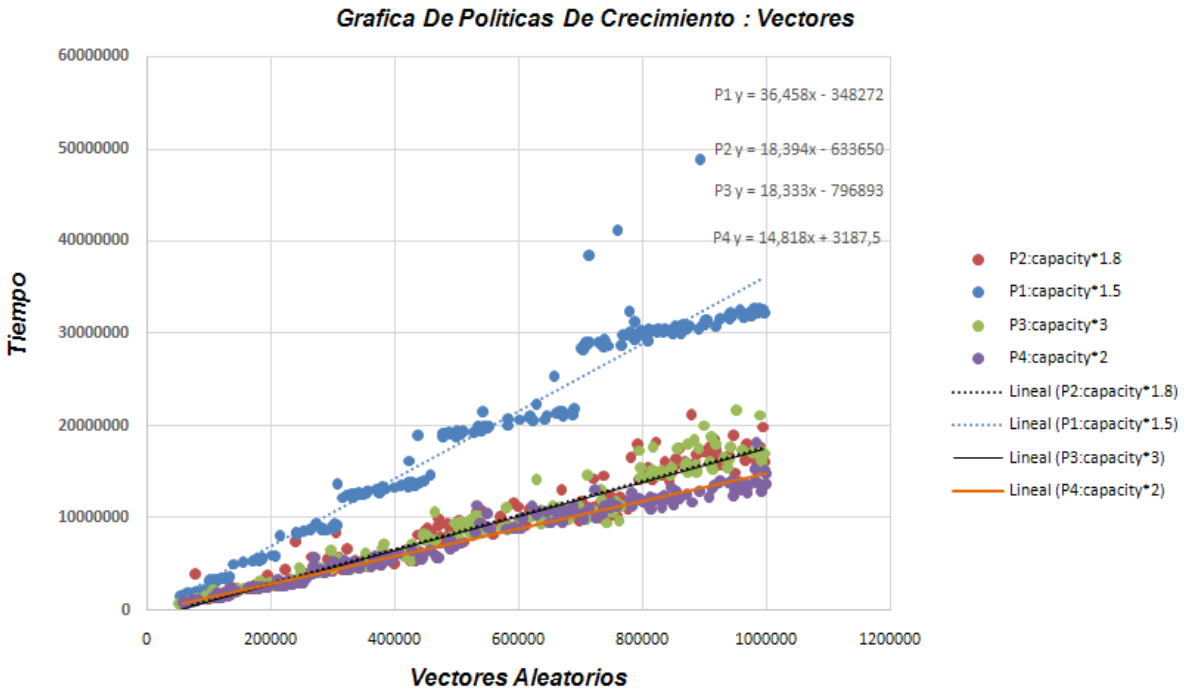


Figure 6: Comparación de Políticas de Crecimiento

En las políticas que probamos en nuestro laboratorio hemos obtenido comportamientos diferentes para cada una de estas, mediante la implementacion realizada en **C++** desciframos como actuaba cada política en particular y como se generaban los datos cuando se llenaba el vector con un dato producido aleatoria mente. generamos un vector que tenia tamaño de 200 elementos aleatorios para cada política.
En nuestras pruebas experimentales cada política de crecimiento varía de acuerdo a la constante multiplicativa que se encuentra en la función **resize()**.

En la gráfica podemos observar que todas las políticas tienen un comportamiento lineal, nosotros realizamos las regresiones lineales correspondientes para obtener las ecuaciones que son las siguientes:

Ecuacion Politica1:	36,458x - 348272
Ecuacion Politica2:	18,394x - 633650
Ecuacion Politica3:	18,333x - 796893
Ecuacion Politica4:	14,818x + 3187,5

Table 2.3: My caption

De acuerdo a la gráfica 6 la política de crecimiento 1 es la que mas tiempo va a tardar en insersion de datos ya que si miramos las pendientes de la tabla 2.3 podemos observar que 36,458x - 348272 la mayor. Si observamos la gráfica, la política número 2 y 3 tienen un comportamiento lineal, ambas se comportan casi de la misma forma, teniendo así un comportamiento parecido de un 93%, nuestro coeficiente de determinación R^2 de estas políticas es del 95%, siendo un coeficiente bastante eficiente.
La diferencia entre estas dos políticas la podríamos observar cuando se está insertando un dato.Para la política 2 se obtuvo una media aritmética en cuanto a tiempo de 9370293.91 ns que es equivalente a 0.9370 s, y para la política 3 una media de 8461568.24 ns que es igual a 0.8461 s,esto podría decirnos que la diferencia entre estas dos políticas en cuanto a tiempo es mínima y es de 0.99 s, lo cual no es muy relevante así sea para la inserción de valores en un vector de mayor tamaño con rangos más elevados.

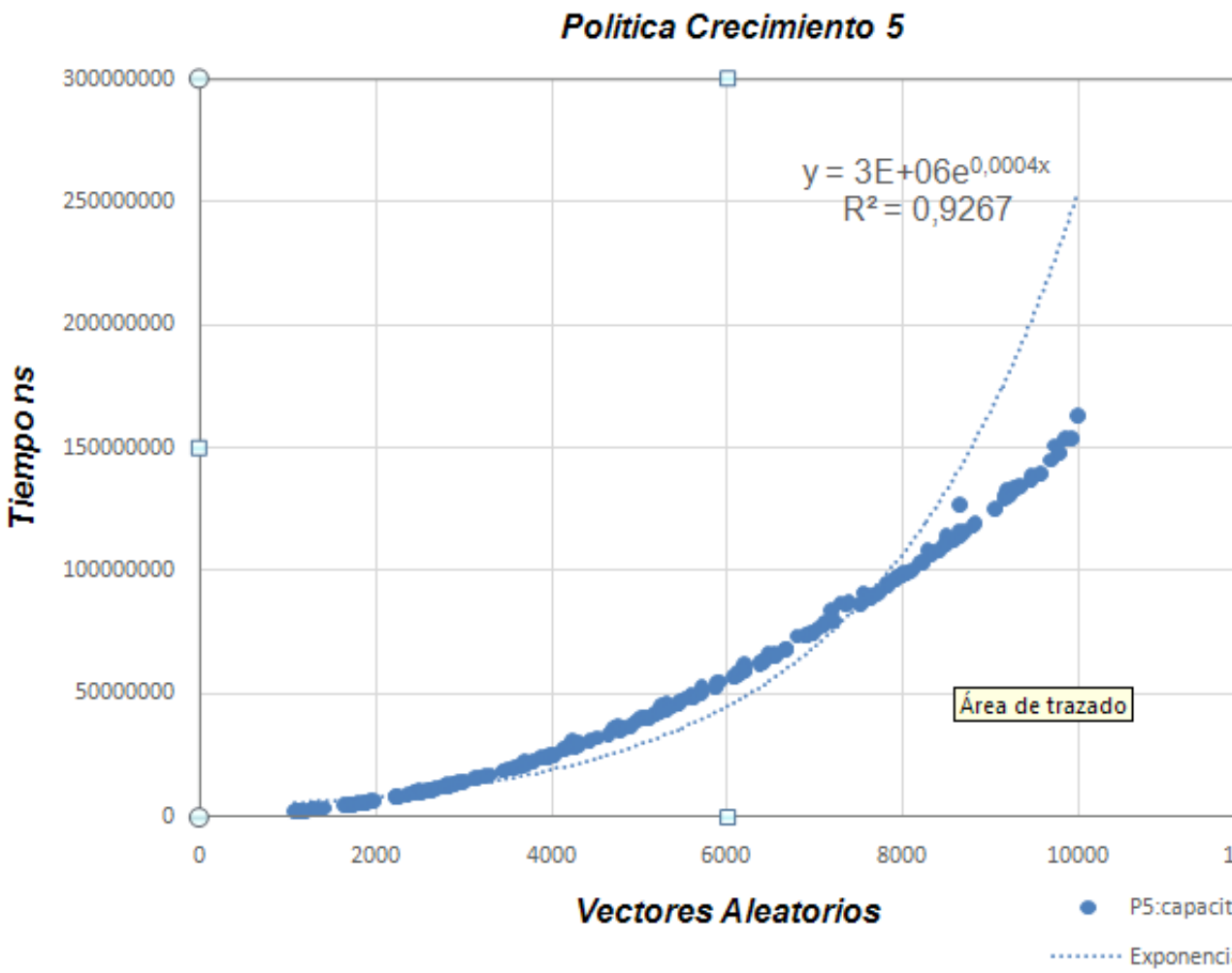


Figure 7: Comparación de Políticas de Crecimiento

En la política 5, que es una política indeseable, podemos observar que su gráfica nos genera un comportamiento exponencial insertando datos, esta gráfica no se pudo anexar a la gráfica de las políticas de crecimiento ya que no podíamos darle los mismos rangos de 50.000 y 1.000.000; en esta política observamos que al generar datos solo usaba el procesador y cuando el algoritmo Produce generaba un dato grande le costaba demasiado tiempo en ejecutarse y arrojar resultados, tanto así que la gráfica que presentamos en este momento tiene rangos desde 1000 hasta 10000, y aun así con rangos tan bajos la gráfica sigue presentando comportamiento exponencial. Esta es la política más ineficiente

de todas, para que esta política se comportara como las anteriores al insertar datos, sería necesario una maquina con un Procesador significativo. La media aritmética de esta política en cuanto a tiempo es de 55597050.1 ns que es igual a 0.555 s, es una cantidad significativa de tiempo porque se está ejecutando con rangos muy bajos, por lo cual, podríamos concluir que si se le diera el mismo rango que a las anteriores tardaría horas en ejecutarse.
El objetivo de este análisis es definir cuál es la política más eficiente para la inserción de datos en el vector, por lo cual hablaremos de la política de crecimiento número 4. Esta política de crecimiento es **nc = capacity*2**, en la gráfica podemos observar que está en la parte inferior marcando los resultados más eficientes en cuanto a tiempo de inserción de datos, esta política tiene una media aritmética en cuanto a tiempo de 7794178.28 ns que es equivalente a 0.779 s, comparándola con las otras se demora mucho menos en insertar datos al vector independientemente de que rangos se le asignen para la elección de datos, los datos generados por esta política tienen un grado de confiabilidad del 97% en su inserción, lo que significa que solo el 3% de sus datos se encuentran dispersos, siendo estala política de crecimiento con una desviación estándar más baja y con un porcentaje de error en los datos más bajo.

2.2 Removiendo Elementos

```
  void recycle() {
  int nc = sz;
  T *ns = new T[nc];
  for (int i = 0; i < sz; i++)
    ns[i] = storage[i];
  delete storage;
  storage = ns;
  capacity = nc;
}
```

.

Código 4: Implementacion de Add

```
//Punto 3b de la tarea, implementacion de remove con eficiencia en memoria,
 //elimina desgaste en memoria dejandola con cero espacios gastados.
  void remove(int i) {
    assert(i >= 0 && i < sz);
    moveOneLeft(i);
     if (sz <= 2 / 3 * capacity)
      recycle();// reciclara el desperdicio dejada por el resize
  }
```

.

Código 5: Implementacion de remove eficiencia de memoria

Esta función que se implemento en el código sirve para remover elementos del vector. Removerá los elementos en la posición que se le asigne, con una parte especial, removerá también el desgaste que se origina cuando se necesita hacer Resize(aumentar capacidad de almacenamiento), de eliminar este desgaste se encarga la función implementada Recycle, esta función es llamada en Remove y allí se encargara de dejar los espacios libres después de que el vector se llene en cero, es decir, no habrá un desgaste de memoria.

2.3 Consumo de Memoria

El experimento que se realizó en este punto para analizar las eficiencias en memorias de las políticas de crecimientos fueron las siguientes:

- Implementar las funciones de Resizes, y Waste.
- Se creó una tabla de datos para analizar todas las políticas.
- Se insertó un dato n igual para todas las políticas para así, entender y analizar bien el funcionamiento de cada una de ellas.
- Se analizó y se interpretó para ese valor de n cada política y como era su eficiencia en cuanto a memoria desperdiciada.

Las siguientes tablas fueron las obtenidas :

			Políticas De Crecimiento				
			P1	**P2**	**P3**	**P4**	**P5**
n = 55000	***Temporal***	***Numero Resizes***	24	17	9	14	54996
capacity = 4	***Espacial***	***Waste***	6447	20465	23732	10536	0

Table 2.4

			Políticas De Crecimiento				
			P1	**P2**	**P3**	**P4**	**P5**
n = 62500	***Temporal***	***Numero Resizes***	25	17	9	14	62496
capacity = 4	***Espacial***	***Waste***	29670	12965	16232	3036	0

Table 2.5

			Políticas De Crecimiento				
			P1	**P2**	**P3**	**P4**	**P5**
n = 80000	***Temporal***	***Numero Resizes***	25	18	10	15	79996
capacity = 4	***Espacial***	***Waste***	12170	55837	156196	51072	0

Table 2.6

			Políticas De Crecimiento				
			P1	**P2**	**P3**	**P4**	**P5**
n = 95000	***Temporal***	***Numero Resizes***	26	18	10	15	94996
capacity = 4	***Espacial***	***Waste***	43255	40837	141196	36072	0

Table 2.7

Analizando el número de resizes que se genera para cada política, vemos que a medida que se aumenta el factor multiplicativo de la política de crecimiento esta genera un número menor de resizes, esto se debe a que cuando se llena el vector empieza a multiplicarse por un valor mayor, lo cual generara más espacio en memoria pero este espacio será llenada por el valor de n que se le asigna al vector. Por ejemplo la política 1 que es capacity*1.5 para el primer valor de n (55000) genera 24 resizes y se puede observar en la tabla que para la política de crecimiento 2 que es capacity*1.8 reduce el número de resizes a 17 y así se hace sucesivamente con la política de crecimiento número 4 que es capacity*2 y al final con la política de crecimiento número 3 que es capacity*3, esta última genera un numero de resizes a 9, si se compara con la primer política es una reducción de 15 resizes, serian 15 resizes menos que se evitaría el programa para llenar el vector con el valor de n asignado para este primer experimento (55000), y esto sucede con cualquier valor de n como se puede observar en la tabla. Tenemos que considerar el caso de la política de crecimiento número 5, que es capacity+1, ya antes dicho en este documento, esta es la política de crecimiento más ineficiente de todas en cuando a inserción y tiempo de ejecución, para este primer experimento con el valor de n (55000) vemos que generar 54996 resizes en su ejecución, esto se debe a que la capacidad inicial de nuestro clase vector es de 4, cuando se está ejecutando este programa por cada dato que se inserte será un resize que el algoritmo deberá ejecutar y el algoritmo tendrá que ejecutar más pasos y el programa tardara mucho tiempo en arrojar resultados.

Analizando el otro parámetro que es Waste o el desperdicio en memoria que ejerce al ejecutar el programa con cada política de crecimiento, se puede observar que la política de crecimiento que ejerce un menor gasto de memoria es la política 1 (capacity+1), esta política no genera ningún desperdicio en memoria, pero al saber que es una política ineficiente en cuestión de tiempo y de numero de resizes a ejecutar, es una política que se debe descartar por completo para nuestra clase vector. Por tanto, nos queda analizar el comportamiento de las otras 4 políticas de crecimiento que tenemos para nuestra clase vector. En cuestión de gasto en memoria, estas políticas de crecimiento no tienen una tendencia común, todo depende de la cantidad de elementos que se deseen insertar en el vector, y de allí, analizar cuál sería la política que ofrece una estabilidad y una relación en tiempo y memoria. En la tabla se puede observar que la política de crecimiento que mejor estabilidad ofrece en memoria es la política numero 2 (capacity*2), es una política de crecimiento efectiva para la inserción de datos en un vector, al inicio del documento pudimos concluir que es la política que mejor ahorro de tiempo ofrece, se obtuvieron resultados estables y efectivos para nuestra clase Vector, para este caso de análisis de desperdicio en memoria es igual, esta política ofrece una cantidad aceptable de resizes y una cantidad de desperdicio de espacios en memoria aceptable,recordando que, en este punto lo que se trata de buscar es una estabilidad para poder describir cual es la política que se debería usar para la inserción de datos de una estructura de datos usando vectores.

3 Anexos

El codigo para esta tarea se encuentra disponible en
"https://github.com/heticor915/DataStructures/blob/master/HomeWorks/Homework2-1/Solution/"