# Arreglos dinámicos

## Introducción

En este taller vamos a estudiar el comportamiento de la estructura de datos `Vector`. Recuerde que esta estructura de datos es una implementación del concepto de *secuencia*. Para este taller usted podrá trabajar en grupos de máximo dos integrantes. Cualquiera de los dos estudiantes podrá ser llamado a sustentar lo presentado y la evaluación de su desempeño podrá ser utilizada como la evaluación de ambos integrantes.

## 1 Qué tan rápida es?

El objetivo de este punto es comparar nuestra estructura de datos tal como la implementamos en clase contra un arreglo convencional. Vamos a considerar $n$ como el número de elementos que va a contener el arreglo y el vector. Adicionalmente vamos a insertar en ambos un único elemento 1. Mediremos el tiempo necesario para almacenar diferentes cantidades de elementos (valores de $n$).

Las siguientes dos funciones almacenan `n` elementos en un arreglo y en un vector. Usted lo que hará es medir el tiempo que tardan ambas funciones en ejecutarse para diferentes valores de `n`.

```
void arrayInsertion(int n) {
  int* a = new int[n];
  for(int i = 0; i < n; i++) {
    a[i] = 1;
  }
}

void vectorInsertion(int n) {
  Vector<int> a;
  for(int i = 0; i < n; i++) {
    a.add(i,1);
  }
}
```

La forma en que produciremos `n` es apartir de un generador de números aleatorios. Considere la siguiente función que retorna un vector de números aleatorios de tamaño `s` con cada número entre `l` y `u`.

```
#include <random>
#include <iostream>

using namespace std;

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;
}
```

Para medir el tiempo utilizaremos un objeto de la clase `Timer`. Para un ejemplo de su uso por favor consulte el archivo `example.cc`.

- Genere una cantidad importante de números. Cada uno de estos será el tamaño de un vector y un arreglo. Mida el tiempo de ejecución de cada llamado a `arrayInsertion` y a `vectorInsertion`. Qué conclusiones puede usted sacar de los resultados de dichas ejecuciones?.

  Cuando se mide el tiempo de ejecución de un programa o una función este puede presentar variaciones. Incluso en ocasiones pueden aparece valores de 0. Para que nuestro experimento (y por ende nuestras conclusiones) no se vean afectadas por esto usted calculará el tiempo varias veces y obtendrá un promedio con el que trabajará.

  Trate de registrar los resultados de sus experimentos en una tabla como la siguiente:

| Tamaño | `arrayInsertion` | `vectorInsertion` |
|---|---|---|
| $t_0$ | | |
| $t_1$ | | |
| $\vdots$ | | |
| $t_n$ | | |

  Cada entrada en la tabla contendrá *el promedio* los tiempos que toman `arrayInsertion` y `vectorInsertion` para la inserción de $t_i$ elementos. Por ejemplo, supongamos

$t_0 = 100$ elementos. El tiempo de `arrayInsertion` se calculará como $T_{100} = (x_1 + x_2 + \cdots + x_k)/k$. Donde $k$ es el número de veces que se repite cada experimento. Es decir, hay que tomar $k$ veces el tiempo que toma la inserción de 100 elementos tanto en el vector como en el arrglo. A estos datos se les saca el promedio y es éste promedio el que se considera en la tabla.

Los valores de tamaño que usted va a considerar no son arbitrarios. En su lugar usted los va a generar aleatoriamente. Es decir, cada $t_i$ será un elemento de un vector que usted generará utilizando la función `produce`. Debe asegurarse de escojer muy bien los valores que pasa a `produce`. Estos *deben* ser significativos. Para el análisis de los datos usted puede utilizar su herramienta favorita.

- Durante la clase vimos que la estructura de datos `Vector` tiene utiliza una expresión para calcular el nuevo tamaño que se utilizará para copiar los elementos del arreglo interno. Considere las siguientes expresiones:
  - `int ns = capacity + 1;`
  - `int ns = capacity * 2;`
  - `int ns = capacity * 3;`
  - `int ns = capacity * 1.8;`
  - `int ns = capacity * 1.5;`

  Realice las modificaciones a la clase para evaluar cada una de las expresiones y su impacto en el tiempo de la función `vectorInsertion`. Explique sus conclusiones, las cuales deben estar justificadas por evidencia matemática del resultado experimental.

## 2 Adicionando elementos

En la sección anterior usted únicamente midió el desempeño cuando se adiciona un elemento al final de la secuencia. Si esto no es evidente para usted por favor revise cuidadosamente la implementación de `vectorInsertion`. De ser necesario, realice un seguimiento o prueba de escritorio para un ejemplo pequeño.

A continuación va a medir en realidad que tan eficiente es la operación `add` en diferentes casos. Para esto usted deberá construir la función `testAdd` que recibe un vector `vec` con una cantidad considerable de datos, digamos $5'000.000$. Su función generará un vector de posiciones válidas en `vec` utilizando la función `produce` del punto anterior y adicionara el número `10` dentro de `vec` en cada una de tales posiciones.

- Qué conclusiones puede sacar de las mediciones realizadas?.
- Cómo puede comparar los resultados obtenidos con la complejidad teórica de la operación `add`?

## 3 Removiendo elementos

Implemente la operación `removeNaive(i)` que toma como argumento una posición y remueve el elemento en el vector en dicha posición. Utilice el mismo procedimiento del punto anterior para tener una idea del consumo de tiempo de la operación. Es decir, cree un vector de una cantidad considerable de elementos (por ejemplo $5'000.000$) y genere posiciones utilizando `produce`. Esta vez usted removerá los elementos en cada una de las posiciones generadas.

- Como en el punto anterior describa sus conclusiones y compare el tiempo obtenido con respecto a la complejidad teórica.

Ahora implemente la operación `remove(i)` que tiene el mismo efecto de `removeNaive` con una pequeña diferencia: eficiencia en memoria. `remove(i)` tendrá la capacidad de reducir el almacenamiento requerido por el vector si su ocupación es menor o igual a $2/3$ de su capacidad. En tal caso, el vector reducirá su desperdicio de memora completamente. Es decir, el arreglo interno deberá tener el número exacto de elementos almacenados en el vector.

## 4 Consumo en memoria

Los ejercicios anteriores sirvieron para darnos una idea de que tan rápida es la estructura de datos `Vector`. Ahora queremos hacer un estudio de que tan eficiente es esta estructura cuando se trata de memoria. Para este estudio usted debe adicionar las siguientes operaciones a la clase `Vector`.

- `waste` que retorna el número de posiciones que el vector tiene reservadas pero que están sin utilizar.
- `numResizes` que retorna el número de veces que el vector ha tenido que ser redimensionado desde su creación.

Estas operaciones *solamente* son relevantes para nuestro estudio y *no* forman parte del conjunto de operaciones real de la estructura de datos.

Con las operaciones anteriores usted ahora evaluará cada una de las expresiones para calcular el nuevo tamaño propuestas en la primera parte. Recuerde que esta vez lo que evaluaremos será el desperdicio de memoria y el número de veces que se ejecuta `resize` en lugar del tiempo. Esta vez será usted quien diseñará el experimento a realizar. Recuerde el objetivo: evaluar el desempeño en espacio de la estructura de datos.

- Describa cuidadosamente el experimento que va a realizar y explique como este conduce conclusiones relevantes de lo que se quiere evaluar.

- Presente los datos y conclusiones de su experimento.
- Evalúe ahora el desempeño en memoria de las dos operaciones para remover elementos propuestas en el punto anterior. Aquí usted tendrá que proponer su experimento, justificarlo y presentar sus conclusiones.