Stats

Introducción

El fin de esta asignación es que aprendas la importancia y uso de estructuras de datos en las cuales la posición en el orden de claves y su acceso rápido es esencial para el desempeño. Para tales efectos, programarás un simple recolector de muestras estadísticas el cual permitirá calcular estadísticas básicas rápidamente, aun para conjuntos muy grandes.

Para la realización de este trabajo necesitarás definir los atributos de tu clase y elegir un tipo de conjunto para almacenar las muestras. Puedes definir tu propio conjunto basado en arreglos, listas enlazadas o árboles binarios de búsqueda en cualquiera de sus versiones. Aun así, si prefieres utilizar algún contenedor de la STL para solucionar tu problema, tienes total libertad para hacerlo.

A continuación se te proveen algunos conceptos importantes para la solución del problema.

Estadísticas básicas

Por simplicidad, para este trabajo nuestras muestras serán números enteros. Las estadísticas a calcular son:

- **Mínimo:** El menor elemento de la muestra
- **Máximo:** El mayor elemento de la muestra
- **Media:** la media aritmética o promedio de los datos de la muestra

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad (1)$$

- **Varianza:**

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \overline{x})^2 \quad (2)$$

- **Primer cuartil:** El elemento situado en la posición infija $\lfloor n/4 \rfloor$, donde n es el tamaño de la muestra
- **Segundo cuartil o mediana:** El elemento situado en la posición $\lfloor n/2 \rfloor$ si el tamaño de la muestra es impar, o $\frac{x_{n/2} + x_{n/2+1}}{2}$ (promedio entre los dos valores del centro) si el tamaño de la muestra es par. En ambos casos, es la posición en secuencia infija
- **Tercer cuartil:** El elemento situado en la posición infija $\lfloor \frac{3}{4} n \rfloor$
- **Tamaño:** Cantidad de elementos en la muestra

Otras operaciones

Aparte de la construcción y copia de una muestra, caracterizada por el tipo **`Sampler`**, la mayoría de las operaciones consisten en extracción mutable o no de submuestras ordenadas; es decir, una muestra subconjunto de otra muestra.

Por ejemplo, el método:

```
std::vector<long> get_by_position_range(size_t l, size_t r) const noexcept
```

Retorna una lista ordenada de las muestras que se encuentran en las posiciones $[l, r)$. Por ejemplo, si la muestra ordenada es `{1, 3, 4, 7, 9, 11, 14, 19}`, entonces `get_by_position_range(2, 5)` retorna `{4, 7, 9}`.

Esquema simple de solución

Un esquema de solución bastante simple, que debes seriamente considerar para tu implementación, consiste en almacenar las muestras desordenadas en un arreglo. De este modo, la inserción de una muestra es sumamente veloz, O(1) y con bajo coste constante.

En este caso, en la mayoría de los cómputos se ordena el arreglo de muestras y, en función de los índices en el arreglo, se efectúan los cómputos necesarios. Por ejemplo, el primer cuartil, la mediana y el tercer cuartil se encontrarán en las posiciones del arreglo n/4, n/2 y 3n/4 respectivamente.

El problema de desempeño con este enfoque, pero no de simplicidad de instrumentación, está dado por la necesidad de repetir el ordenamiento si la muestra se modifica: se añaden o eliminan elementos. En este caso, el coste por ordenar puede ser bastante severo, especialmente si la muestra se modifica muy a menudo.

Esquema de solución de alto desempeño

El esquema de alto desempeño y escala subyace en el empleo de árboles binarios de búsqueda equilibrados con rangos para almacenar los elementos de la muestra. En este caso, la inserción tiende a O(lg n), pero no se requiere ordenar el arreglo. El acceso por posición de ordenamiento proporciona un coste O(lg n), que es el coste que se pagará por obtener el primer cuartil, la mediana y el tercer cuartil respectivamente. Además, la extracción de un subconjunto de tamaño m podrá hacerse con complejidad O(lg n), en lugar de O(n lg n) + O(m) que se requerirá si la muestra se almacena en un arreglo.

El problema con este enfoque es doble. En primer lugar, el uso de árboles solo se compensa con escalas de muestras muy grandes, del orden de los millones de elementos; con escalas

menores el arreglo es tan o más rápido. En segundo lugar, la instrumentación con árboles es más compleja, especialmente para la extracción de subconjuntos.

Por supuesto, un enfoque intermedio es perfectamente permitido. Por ejemplo, podrás usar árboles binarios de búsqueda más rápidos, pero ejecutar las operaciones de extracción por su recorrido infijo.

Dilema instrumental

Uno de los fines de que en este laboratorio el desempeño tenga una ponderación importante, es presentarte y forzarte a confrontar con un dilema ingenieril que es más frecuente de lo que se podría presumir: ¿debo hacer una implementación simple y probablemente más segura, o una de alto desempeño y escala, pero más propensa a errores y más difícil de depurar?

Si elijes utilizar árboles binarios de búsqueda, ten en cuenta que tus algoritmos deberán soportar duplicados, para esto, podrías usar un comparador que, en lugar de usar “menor que”, use “menor o igual que”. Pondera en tu dilema que si hay muchas repeticiones entonces el desempeño de la inserción en un árbol binario de búsqueda puede degradarse.

La práctica

Para la realización de este trabajo, se te provee un proyecto en C++ con la siguiente estructura:

```
CC03
├── lib
│   ├── include
│   │   ├── BasicStats.hpp
│   │   ├── IOUtilities.hpp
│   │   ├── Sampler.hpp
│   │   ├── TestUtilities.hpp
│   │   └── Types.hpp
│   ├── obj
│   └── src
│       ├── BasicStats.cpp
│       ├── IOUtilities.cpp
│       ├── Sampler.cpp
│       ├── TestUtilities.cpp
│       └── Types.cpp
├── Makefile
└── test
    ├── bin
    └── src
        └── TestSampler.cpp
```

El código fuente

`IOUtilities.hpp` y `IOUtilities.cpp` contienen algunos operadores para imprimir por la salida estándar arreglos, secuencias, pares ordenados y estadísticas básicas. Además, provee la macro `LOG`, la cual funciona como la función `printf` y permite imprimir mensajes con formato que serán mostrados solo cuando se compila en modo **debug**. Un ejemplo de uso es: `LOG("El valor de la variable x es %d", x)` suponiendo que x es una variable de tipo `int`. La intención de la macro es que puedas usarla para imprimir mensajes que requieras sin contaminar la salida del evaluador. Estudia estos archivos y no los modifiques.

`TestUtilities.hpp` y `TestUtilities.cpp` contienen funciones utilitarias para comparar secuencias, pares ordenados y estadísticas básicas. Además provee la función `assert_equal` para ser usada en los casos de prueba. Estudia estos archivos y no los modifiques.

`Types.hpp` y `Types.cpp` contienen la definición del alias `SPair` como una tupla de dos enteros. Estudia estos archivos y no los modifiques.

`BasicStats.hpp` y `BasicStats.cpp` contienen la definición de una estructura para almacenar las estadísticas básicas. Estudia estos archivos y no los modifiques.

`test/src/TestSampler.cpp` es un programa para ejecutar los casos de prueba. Solo están algunos. Queda a ti agregar los que faltan.

`Makefile` contiene las reglas para compilar.

Lo que debes resolver

Tu trabajo consiste en implementar la clase `Sampler` que se encuentra en los archivos `lib/include/Sampler.hpp` y `lib/src/Sampler.cpp`.

Constructores

Implementa el constructor por omisión y el constructor de copia.

```
Sampler() noexcept;
Sampler(const Sampler& s) noexcept;
```

Swap

Implementa el método

void `swap(Sampler& sampler) noexcept;`

El cual debe intercambiar, en O(1), el contenido de `this` con `sampler`.

Tamaño de la muestra

Programa el método

```
size_t size() const noexcept;
```

El cual debe retornar la cantidad de elementos en la muestra.

Media

Programa el método

```
double mean() const noexcept;
```

El cual retorna la media aritmética de la muestra. **Usa la ecuación (1)**. Si la muestra está vacía, se debe arrojar la excepción **`std::length_error`** con el mensaje **`"empty sampler"`**.

Varianza

Programa el método

```
double variance() const noexcept;
```

El cual retorna la varianza de la muestra. **Usa la ecuación (2)**. Si la muestra está vacía, se debe arrojar la excepción **`std::length_error`** con el mensaje **`"empty sampler"`**.

Añadidura de elemento

Programa el método

```
void add_sample(long sample) noexcept;
```

El cual añade el valor de `sample` a la muestra.

Obtención de elemento

Programa el método

```
long get_sample(size_t i) const noexcept;
```

El cual retorna el i-ésimo menor elemento de la muestra. Si `i` es mayor o igual que el tamaño de la muestra, se debe arrojar la excepción **`std::out_of_range`** con el mensaje **"index out of range"**

Lista de muestras

Programa el método

```
std::vector<long> list() const noexcept;
```

El cual retorna una lista ordenada con todos los elementos de la muestra.

Estadísticas básicas

Programa el método

```
BasicStats stats() const noexcept;
```

El cual retorna las estadísticas básicas de la muestra en un objeto de tipo `BasicStats`. Si la muestra está vacía, se debe arrojar la excepción **`std::length_error`** con el mensaje **`"empty sampler"`**.

Lista por rango de posiciones

Programa el método

```
std::vector<long> get_by_position_range(size_t, size_t r) const;
```

El cual retorna una lista de los elementos comprendidos en el rango **`[l, r)`**. Atención, cualquier rango **`[i, i)`** es válido y consiste en una lista con el elemento **`i`**.

Si `l` es mayor que `r`, entonces se debe arrojar la excepción **`std::invalid_argument`** con el mensaje **“crossed range”**. Si alguno de los índices es mayor o igual que el tamaño de la muestra se debe arrojar la excepción **`std::out_of_range`** con el mensaje **“index out of range”**.

Es propicio el momento para introducir una ligera dificultad, la cual sólo se te presentará si optas por una implementación de alto desempeño basada en árboles con rangos. En este caso, podrías extraer el rango deseado mediante una partición por posición. No obstante, y he aquí la dificultad, el método está declarado como inmutable (el cualificador `const`). Para poder extraer el rango mediante esta estrategia es necesario modificar temporalmente el objeto y después restaurarlo a su estado original. En C++, y es probable que sea uno de los pocos lenguajes donde esto está permitido, puedes, a tu cuenta, riesgo y responsabilidad, cambiar temporalmente el carácter constante de un objeto mediante un `const_cast` y modificar el objeto. El compromiso es que restaures el objeto al estado original antes de dar el resultado, pues si no, las premisas de generación de código objeto serán comprometidas.

Aparte de informar al compilador, la rutina se declara `const` porque no modifica al objeto. La lista resultante es una copia.

Una alternativa es que tú elimines el cualificador const de la declaración. Este enfoque, aunque más sincero, plantea dos problemas potenciales. El primero es que al asumir que el objeto es modificable el compilador podría dejar de efectuar algunas optimizaciones. El segundo es que, aunque el evaluador no verifica explícitamente por el cualificador, éste podría no compilar. Es a tu riesgo si optas por esta alternativa.

Lista de elementos por rango de valores

Programa el método

```
std::vector<long> get_by_key_range(long sl, long sr) const;
```

El cual retorna una lista ordenada de las muestras que son mayores que `sl` y menores o iguales que `sr`.

Nota, y ten mucho cuidado con una eventual confusión, que los valores `sl` y `sr` **no son posiciones respecto a la secuencia ordenada**, tal como en el método anterior.

Si `sl` es mayor que `sr`, entonces se debe arrojar la excepción **`std::invalid_argument`** con el mensaje **“crossed keys”**.

Obtención de elementos por lista

Programa el método

```
std::vector<SPair> get_by_keys(const std::vector<long>& keys) const noexcept;
```

En este caso recibe como entrada una lista de valores a buscar dentro de la muestra. El método retorna una lista de pares donde el primer elemento es el valor buscado y el segundo la posición (en secuencia ordenada) del valor dentro de la muestra. Si el elemento no se encuentra en la muestra, entonces la posición debe ser **-1**. La lista resultante debe estar ordenada por el valor de la muestra.

Corte por rango de valores

Programa el método

```
Sampler& cut_by_key_range(long sl, long sr);
```

El cual elimina de `this` todos los elementos que son menores o iguales que `sl` y mayores que `sr`.

Si `sl` es mayor que `sr`, entonces se debe arrojar la excepción **`std::invalid_argument`** con el mensaje **`"crossed keys"`**.

Corte por rango de posiciones

Programa el método

```
Sampler& cut_by_position_range(size_t l, size_t r);
```

El cual elimina de `this` todos los elementos fuera del intervalo **`[l, r)`**

Si `l` es mayor que `r`, entonces se debe arrojar la excepción **`std::invalid_argument`** con el mensaje **`"crossed range"`**. Si alguno de los índices es mayor o igual que el tamaño de la muestra se debe arrojar la excepción **`std::out_of_range`** con el mensaje **`"index out of range"`**.

Evaluación

- La práctica debe ser resuelta individualmente. Se penalizará por plagio.
- Puedes realizar preguntas por el chat de la práctica si no implica compartir código.
- Tienes ocho (8) días para enviar la solución.
- Puedes entregar tres (3) veces.
- Solo puedes hacer una entrega por día.
- El estilo de código tiene un peso del 10% de la calificación de la evaluación y se evaluará la velocidad de ejecución.
- Los archivos que entregues deben tener tu nombre, apellido y cédula como comentario al inicio.
- No se permite usar ningún statement de impresión por salida estándar. Para eso, usa la macro `LOG`.
- Si no cumples con las reglas anteriores, no será procesada la evaluación y perderás una entrega.

Para poder evaluarte, debes enviar un correo solamente con los archivos **`Sampler.hpp`** y **`Sampler.cpp`** **(NO COMPRIMIDOS)** como adjunto a la dirección alejandro.j.mujic4@gmail.com con el asunto: **[PR3]-03-Stats**. El cuerpo del correo debe tener tu nombre y número de cédula.

Deadline para entregas: Viernes 11/04/2025 23:59:59 VET