Tarea de implementación

9 de octubre de 2018

Índice

1. Objetivo

El objetivo de esta tarea es el modelado y posterior implementación de un problema utilizando las técnicas Greedy, Programación Dinámica y Divide and Conquer.
Se espera que la implementación realizada haga un buen uso de la memoria.

2. Problema

Se considera un conjunto de n pedidos $\{1, \ldots, n\}$, de los cuales se conoce su tiempo de inicio, su tiempo de fin y su volumen.
Dado un pedido i, se denota su tiempo de inicio como $inicio(i)$, su tiempo de fin como $fin(i)$ y su volumen como $volumen(i)$.
Se considera que dos pedidos $i,\ j \in \{1, \ldots, n\}$ son *compatibles* **si y solo si** no se superponen en el tiempo, es decir:

$$compatibles(i,j) = \begin{cases} true & \text{si } fin(i) \leq inicio(j) \\ & \vee\ fin(j) \leq inicio(i), \\ false & \text{en otro caso.} \end{cases}$$

Se pide:

a) Una solución, basada en la técnica Greedy, al problema de obtener el conjunto de pedidos compatibles de cantidad máxima. Esta solución deberá implementarse en la función `max_cantidad` que deberá ejecutarse en $O(n\ log\ n)$.

b) Una solución, basada en la técnica Programación Dinámica, al problema de obtener el conjunto de pedidos compatibles que maximice la suma de sus volúmenes. Esta solución deberá implementarse en la función `max_volumen` que deberá ejecutarse en $O(n\ log\ n)$.

3. Materiales

El material se encuentra en el archivo `MaterialesTareaImplementacion.tar.gz` que está disponible en la sección **Laboratorio** del sitio EVA del curso. Se desempaqueta usando el comando `tar`:

```
$ tar zxvf MaterialesTareaImplementacion.tar.gz
```

Se obtienen los siguientes archivos:

- El intérprete de comandos `principal.cpp` utilizado para testear la implementación (ver sección 8.1).
- El archivo `Makefile` con reglas para la compilación y testing (ver sección 8.2).
- En el subdirectorio `include` el archivo de definición: `intervalos.h`.
- En el subdirectorio `test` los casos de prueba para `principal.cpp`.

4. ¿Qué se pide?

Se debe implementar el módulo `intervalos.cpp` que implementa las funciones declaradas en `intervalos.h`.

5. Entrega

Se debe entregar el archivo `intervalos.cpp`.

5.1. Plazo

La entrega se podrá realizar hasta el **lunes 5/11** a las **13:00hs**.

5.2. Corrección

La corrección se hará en las máquinas de Facultad, con algunos de los casos que se hayan publicado y con otros que se publicarán después de terminado el plazo de entrega. Se recomienda verificar la entrega en las máquinas Linux de Facultad previo a la entrega. La verificación se puede hacer tanto de forma presencial como remota conectándose vía ssh.
Como se mencionó en la Sección 1 se espera que la implementación realizada haga un buen uso de la memoria por lo que la corrección se realizará utilizando el analizador `valgrind`.
A las personas que realicen una entrega considerada insatisfactoria se les permitirá hacer una re-entrega. El procedimiento para la re-entrega será el mismo que para la entrega original. El plazo para la re-entrega será de 24 horas después de comunicados los resultados de la entrega. En la re-entrega la exigencia será mayor que en la entrega original.

5.3. Identificación

El archivo a entregar debe tener en la primera línea un comentario con la cédula **sin guión ni dígito verificador**. Ejemplo:

```
/* 1234567 */
```

6. Lenguaje y sistema operativo

Sistema operativo

El sistema operativo es la versión de `Linux` instalada en las máquinas de la facultad.

Lenguaje

El lenguaje que se utiliza es `C*`, que es una extensión de `C` con las siguientes funcionalidades de `C++`:

- Funciones `new` y `delete`.
- Pasaje por referencia.
- Tipo `bool`.
- Definiciones de variables de tipo `struct` o `enum` al estilo `C++`.

Herramientas

Las herramientas que se deben tener instaladas para poder desempaquetar, compilar, ejecutar y testear el laboratorio son: `tar`, `gzip`, `g++`, `make`, `valgrind` y `diff`.

Compilador

El compilador es la versión de `g++` instalada en las máquinas linux de la facultad.

7. Individualidad

En la tarea de implementación se debe cumplir el Reglamento sobre no individualidad que está en la sección `Laboratorio` del sitio `web` del curso.

8. Apéndice

8.1. Verificación

La implementación de código es solo una parte del desarrollo de software. Tras la implementación de cada módulo, de cada función, se debe verificar que se está cumpliendo con los requerimientos.

Se incluye el módulo `principal.cpp` que funcionará como verificador del módulo `intervalos`. El mismo lee desde la entrada estándar, procesa lo leído y escribe en la salida estándar. Se incluyen como ejemplos algunos casos de prueba (que **NO** necesariamente serán los únicos utilizados para la corrección). En ellos cada archivo `.in` representa lo leído en la entrada estándar y el correspondiente archivo `.out` lo que se escribe en la salida estándar. Se pueden redireccionar la entrada y salida estándar con los operadores de redirección < y > respectivamente. Por lo tanto se puede ejecutar `principal` con cada archivo `.in` redirigiendo la salida hacia otro archivo (por ejemplo, con extensión `sal`) que luego se compara con el archivo `.out` correspondiente. La comparación se hace con la utilidad `diff`. Si los archivos comaparados son iguales no se imprime nada en la salida, por lo que si la salida de la ejecución de `diff` se redirige hacia un archivo, este tendrá tamaño 0. Se muestra un ejemplo de ejecución y comparación exitosa:

```
$ ./principal < test/01.in > test/01.sal
$ diff test/01.out test/01.sal > test/01.diff
```

8.2. Makefile

Para automatizar el proceso de desarrollo se entrega el archivo `Makefile` que consiste en un conjunto de reglas para la utilidad `make`.
Cada regla consiste en un *objetivo*, las *acciones* para conseguir el objetivo y las *dependencias* del objetivo. Cuando el objetivo y las dependencias son archivos, las acciones se ejecutan cuando el objetivo no está actualizado respecto a las dependencias (o sea, es un archivo que no existe o su fecha de modificación es anterior a la de alguna de las dependencias). Por más información ver el manual de `make` y el Instructivo Makefile que está en la Sección **Laboratorio** del sitio `web` del curso.
En el `Makefile` entregado las reglas incluidas son:

- `principal`: para compilar y enlazar.
- `clean`, `clean_bin` y `clean_test`: para borrar archivos.
- `testing`: para hacer pruebas.

make principal La regla `principal` compila y luego genera el ejecutable `principal`. Esta regla es la predeterminada, o sea que es la que se invoca si no se especifica ninguna.

En la siguiente secuencia se ve una ejecución exitosa de `make` junto con el estado de los directorios antes y después.

```
$ ls
include intervalos.cpp Makefile  principal.cpp test

$ make
g++ -Wall -Werror -I -g -DNDEBUG principal.cpp intervalos.cpp -o principal

$ ls
include intervalos.cpp Makefile principal principal.cpp test
```

Si no se hacen cambios y se vuelve a correr `make` no se hace nada.

```
$ make
make: No se hace nada para «all».
```

Si hay errores de compilación se muestran en la salida estándar.

```
$ make
g++ -Wall -Werror -I -g -DNDEBUG principal.cpp intervalos.cpp -o principal
intervalos.cpp: In function ‘bool* max_cantidad(const intervalo_t*, uint)’:
intervalos.cpp:77:3: error: no return statement in function returning non-void [-Werror=return-type]
   }
   ^
intervalos.cpp: At global scope:
intervalos.cpp:79:3: error: expected unqualified-id before ‘delete’
   delete [] ordenados;
   ^
intervalos.cpp:80:3: error: expected unqualified-id before ‘return’
   return result;
   ^
intervalos.cpp:81:1: error: expected declaration before ‘}’ token
 } // max_cantidad
 ^
cc1plus: all warnings being treated as errors
Makefile:45: fallo en las instrucciones para el objetivo ’principal’
make: *** [principal] Error 1
```

make testing Con la regla `testing` se ejecuta el programa con los casos de entrada (`in`) generando archivos con la extensión `sal` y estos se comparan con las salidas esperadas (`out`), obteniendo archivos con extensión `diff`.
El hecho de que en la salida no haya referencias al proceso de ejecución y comparación indica que las comparaciones fueron exitosas.

```
$ make testing
./principal < test/00.in > test/00.sal
./principal < test/01.in > test/01.sal
./principal < test/02.in > test/02.sal
./principal < test/03.in > test/03.sal
./principal < test/04.in > test/04.sal
./principal < test/05.in > test/05.sal
```

Una nueva ejecución no hará nada:

```
$ make testing
```

Si los archivos a comparar no son iguales se indica en la salida estándar. Y se puede comprobar que en el directorio `test` los archivos `diff` no tienen tamaño 0.

```
$ make testing
./principal < test/00.in > test/00.sal
---- ERROR en caso test/01.diff ----
./principal < test/02.in > test/02.sal
./principal < test/03.in > test/03.sal
---- ERROR en caso test/03.diff ----
./principal < test/04.in > test/04.sal
```

```
./principal < test/05.in > test/05.sal
-- CASOS CON ERRORES --
03
01

$ stat --print="%n  %s \n" test/*.diff
%test/00.diff  0
%test/01.diff  59
%test/02.diff  0
%test/03.diff  143
%test/04.diff  0
%test/05.diff  0
%
```

Observación: la ejecución anterior es una simplificación de la ejecución completa que se muestra a continuación.

```
$timeout 6 valgrind -q --leak-check=full ./principal < test/00.in > test/00.sal 2>&1
```

Donde `timeout 6` significa que la ejecución del caso se cortaría en caso de demorar más de 6 segundos y `valgrind -q -leak-check=full` es la aplicación que analiza la correcta utilización de la memoria.

make clean En ocasiones puede ser útil borrar los archivos generados. Esto se hace con `make clean`. Si sólo se desea borrar los archivos generados por la compilación se debe invocar `make clean_bin`. Para borrar sólo los archivos generados por la ejecución de `principal` se debe invocar `make clean_test`.

8.3. assert

Para realizar diagnósticos durante el proceso de desarrollo se puede usar la macro `assert`, que toma como parámetro una expresión booleana. Si el resultado de la expresión es `true` no hace nada, pero si el resultado es `false` termina la ejecución del programa indicando el error. Se puede usar para verificar el cumplimento de pre o post condiciones, de invariantes en un bucle, etc. Para usarla se debe incluir el archivo de encabezamientos correspondiente:

```
#include <assert.h>
```

Ejemplo: Con

```
  int cant_scanf = scanf("%s", nom_com);
  assert(cant_scanf == 1);
```

se controla que la lectura desde la entrada estándar asigne exactamente un valor a una variable.
El uso de `assert` debe limitarse a la etapa de desarrollo porque enlentece la ejecución del programa. Por ejemplo, si se quiere comprobar que se cumple una precondición de pertenencia a una lista se puede incluir:

```
  assert(pertenece_a_lista(num, lst));
```

Pero esto implica una recorrida a una lista, que no es necesaria, ya que debe asumirse como precondición.
Para evitar tener que remover del código todas las instancias de `assert` se dispone de la directiva `DNDEBUG` como opción de compilación. Su uso se puede ver en la línea 54 del `Makefile`.

Precaución. La expresión pasada como parámetro a `assert` no debe tener efectos secundarios, ya que estos no se producirán cuando se compile con la directiva `DNDEBUG`. La siguiente invocación será removida del código al incluir esta directiva al compilar:

```
  assert(scanf("%s", nom_com));
```

Por lo tanto se remueve la lectura y asignación del valor de la variable. La forma correcta de uso es la que está al inicio de esta sección.