Algoritmos y Estructuras de Datos
Guía de Trabajos Prácticos Nro. 1

[Las guías deberán ser presentadas en un archivo .cpp con todas las funciones pedidas. Sólo se aceptaran éstas a través de la plataforma. En cada algoritmo desarrollado debe especificarse el orden del mismo. En clase se elegirán alumnos al azar para que expliquen como han resuelto los ejercicios]

1. Escribir una función ***void sort(list<int> &L)***, que ordena los elementos de L de menor a mayor. Para ello emplear el siguiente algoritmo simple, utilizando una lista auxiliar L2: tomar el menor elemento de L, eliminarlo de L e insertarlo al final de L2 hasta que L este vacía. Luego insertar los elementos de L2 en L.

2. Escribir una función ***void selection_sort(list<int> &L)***, que ordena los elementos de L de menor a mayor. Esta función debe ser IN PLACE. Para ello debe tomarse el menor elemento de L e intercambiarlo (swap) con el primer elemento de la lista. Luego intercambiar el menor elemento de la lista restante, con el segundo elemento, y así sucesivamente. Este algoritmo se denomina "Selection Sort" (http://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n). Para desarrollar este algoritmo investigar la función *swap* que se encuentra en la librería *algorithm* de la *STL* (http://www.cplusplus.com/reference/algorithm/swap/).

3. Escribir una función ***void concat_lists(list<list<int> > &LL, list<int> &L)***, que concatena las n listas de LL en L.

4. Escribir una función ***void invert(list<int> &L)***, que invierte el orden de la lista L. Este algoritmo debe implementarse IN PLACE y debe ser O(n).

5. Escribir una función ***void junta(list<int> &L, int c)*** que, dada una lista L, agrupa de a c elementos, dejando su suma IN PLACE. Por ejemplo, si se le pasa como argumento la lista L=(1,3,2,4,5,2,2,3,5,7,4,3,2,2), después de aplicar el algoritmo junta(L,3) debe quedar L(6,11,10,14,4). El algoritmo debe tener un tiempo de ejecución O(n).

6. Dada una lista de enteros L y dos listas SEQ y REEMP (posiblemente de distintas longitudes), escribir una función ***void reemplaza(list<int> &L, list<int> &SEQ, list<int> &REEMP)***, que busca todas las secuencias de SEQ en L y las reemplaza por REEMP. Por

ejemplo, si L=(1,2,3,4,5,1,2,3,4,5,1,2,3,4,5), SEQ=(4,5,1) y REEMP=(9,7,3), entonces después de llamar a reemplaza(L,SEQ,REEMP), debe quedar L=(1,2,3,9,7,3,2,3,9,7,3,2,3,4,5). Para implementar este algoritmo primero buscar desde el principio la secuencia SEQ, al encontrarla, reemplazar por REEMP, luego seguir buscando a partir del siguiente elemento al último de REEMP.

7. **Problema de Josephus.** Un grupo de soldados se haya rodeado por una fuerza enemiga. No hay esperanzas de victoria si no llegan refuerzos y existe solamente un caballo disponible para el escape. Los soldados se ponen de acuerdo en un pacto para determinar cuál de ellos debe escapar y solicitar ayuda. Forman un círculo y se escoge un número *n* al azar. Igualmente se escoge el nombre de un soldado. Comenzando por el soldado cuyo nombre se ha seleccionado, comienzan a contar en la dirección del reloj alrededor del círculo. Cuando la cuenta alcanza el valor *n*, este soldado es retirado del círculo y la cuenta comienza de nuevo, con el soldado siguiente. El proceso continúa de tal manera que cada vez que se llega al valor de *n* se retira un soldado. El último soldado que queda es el que debe tomar el caballo y escapar. Entonces, dados un número *n* y una lista de nombres, que es el ordenamiento en el sentido de las agujas del reloj de los soldados en el círculo (comenzando por aquél a partir del cual se inicia la cuenta), escribir un procedimiento que obtenga los nombres de los soldados en el orden que han de ser eliminados y el nombre del soldado que escapa.

8. Escribir una función ***void sort(list<int> &L)***, que ordena los elementos de L de mayor a menor. Para ello emplear el siguiente algoritmo simple, utilizando una pila auxiliar P: ir tomando el menor elemento de L, eliminarlo de L e insertarlo en P hasta que L este vacía. Luego insertar los elementos de P en L.

9. Escribir una función ***void sort(list<int> &L)***, que ordena los elementos de L de menor a mayor. Para ello utilizar el siguiente algoritmo simple, utilizando una cola auxiliar C: ir tomando el menor elemento de L, eliminarlo de L e insertarlo en C hasta que L este vacía. Luego insertar los elementos de C en L.

10. Escribir una función que reciba como parámetro una cadena de texto y determine si ésta es un palíndromo, ignorando los espacios entre palabras. Un palíndromo es una secuencia de caracteres que se lee igual hacia adelante que hacia atrás, por ejemplo: a*lli si maria avisa y asi va a ir a mi silla*. Recordar que un string puede indexarse como un vector. Con el fin de utilizar la estructura <*list*>, primero deben pasarse los elementos del string a una lista y solo utilizar ésta en el algoritmo.

11. Dos correspondencias M1 y M2 son inversas una de la otra si tienen el mismo número de asignaciones y para cada par de asignación $x \to y$ en M1 existe el par $y \to x$ en M2. Escribir una función ***bool areinverse(map<int,int> &M1, map<int,int> &M2)***; que determina si las correspondencias M1 y M2 son inversas.

12. Escribir una función ***void map2list(map<int,int> &M, list<int> &Keys, list<int> &Vals)***; que dado un map M retorna las listas de claves y valores. Ejemplo: si M = { $1 \to 2$, $3 \to 5$, $8 \to 20$ }, entonces debe retornar Keys = (1,3,8) y Vals = (2,5,20).

13. Escribir una función ***void list2map(map<int,int> &M, list<int> &Keys, list<int> &Vals)***; que dadas las listas de claves (k1,k2,k3, ...) y valores (v1,v2,v3, ...) retorna el map M con las asignaciones correspondientes { $k1 \to v1$, $k2 \to v2$, $k3 \to v3$, ...}. Nota: si hay claves repetidas, sólo debe quedar la asignación correspondiente a la última clave en la lista. Si hay menos valores que claves utilizar cero como valor. Si hay más valores que claves, ignorarlos.

14. Dadas dos correspondencias A y B, que asocian enteros con listas ORDENADAS de enteros, escribir una función ***void merge_map(map<int, list<int> > &A, map<int, list<int> < &B, map<int, list<int> > &C)*** que devuelve en C una correspondencia que asigna el al elemento x la fusión ORDENADA de las dos listas A[x] y B [x]. Si x no es clave de A, entonces C[x] debe ser B[x] y viceversa. Sugerencia: implementar y utilizar una función ***merge(list<int> &L1, list<int> &L2, list<int> &L)***; que devuelve en L la fusión ordenada de L1 y L2. Por ejemplo:

$$A = \begin{Bmatrix} 1 \to (2,3) \\ 3 \to (5,7,10) \end{Bmatrix} \qquad B = \begin{Bmatrix} 3 \to (7,8,9,11) \\ 5 \to (7,10) \end{Bmatrix} \qquad B = \begin{Bmatrix} 1 \to (2,3) \\ 3 \to (5,7,7,8,9,10,11) \\ 5 \to (7,10) \end{Bmatrix}$$

15. Implemente una función ***void cutoffmap(map<int, list<int> > &M, int p, int q)***; que elimina todas las claves que NO están en el rango [p,q). En las asignaciones que quedan también debe eliminar los elementos de la lista que no están en el rango. Si la lista queda vacía entonces la asignación debe ser eliminada. Por ejemplo: si M = { $1 \to (2,3,4)$, $5 \to (6,7,8)$, $8 \to (4,5)$, $3 \to (1,3,7)$ }, entonces *cutoffmap(M,1,6)* debe dejar M = { $1 \to (2,3,4)$, $3 \to (1,3)$ }. Notar que la clave 5 ha sido eliminada si bien está dentro del rango porque su lista quedaría vacía. Restricciones: el programa no debe usar contenedores auxiliares.