Solución Primer Parcial de Programación 3 2012 (28/09/2012)

Instituto de Computación, Facultad de Ingeniería

Ejercicio 1

Parte a)

La ciudad se puede modelar con un grafo no dirigido donde los vértices representan las zonas y las aristas representan las adyacencias entre zonas limítrofes. El problema dado implica encontrar la distancia desde cada uno de los dos vértices dados (*idZona1* e *idZona2*) a todos los demás. Luego alcanza con observar y separar aquellos que están a la misma distancia de ambos vértices.

Dado que la distancia entre dos zonas se define como el largo del camino más corto entre ellas en términos de adyacencia, el requerimiento puede resolverse tomando como base la recorrida *BFS* del grafo. Para cada zona donde se localizan los centros de envío en cuestión (*idZona1* e *idZona2*) se comienza un recorrido BFS que permite medir la distancia desde dichas zonas a todas las restantes.

Luego para cada zona se comparan las dos distancias a las que se encuentra desde las dos zonas en cuestión. Las zonas cuyas distancias coinciden pertenecen a la lista de zonas equidistantes requerida.

Parte b)

```
#define N …

void RecorridaBFS(Grafo g, int idZona, int *nivel) {
      ListaEnteros Cola = CrearLista();   // Cola de zonas
      int procesado[N] = {0};

      ColaInsertarUltimo(Cola, idZona);
      nivel[idZona] = 0;

      while ( ! ListaEsVacia(Cola)) {
            int idCom;  // Identificador de zona de comienzo
            ListaEnteros Ady;  // Zonas adyacentes a una zona

            idCom = ListaPrimero(Cola);
ListaSacarPrimero(Cola);
            Ady = AdyacentesNodoGrafo(g, idCom);
      while ( ! ListaEsVacia(Ady)) {
      int idCorr = ListaPrimero(Ady);
      ListaSacarPrimero(Ady);
      if ( ! procesado[idCorr]) {
                        procesado[idCorr] = 1;
      ListaInsertarUltimo(Cola, idCorr);
      nivel[idCorr] = nivel[idCom] + 1;
      }
            }
      ListaDestruir(Ady);
      }
      ListaDestruir(Cola);
}
```

```
ListaEnteros*  ZonasFrontera(Grafo g, int idZona1, int idZona2) {
      ListaEnteros Zonas  = CrearLista();
      int nivel1[N] = {-1};   // Distancia de cada zona desde idZona1
      int nivel2[N] = {-1};   // Distancia de cada zona desde idZona2

      RecorridaBFS(g, idZona1, nivel1);
      RecorridaBFS(g, idZona2, nivel2);
      for (int i = 0; i < N; i++)
            if (nivel1[i] <> -1 && nivel2[i] <> -1 &&
                nivel1[i] == nivel2[i])
                  ListaInsertarUltimo(Zonas, i);

      return Zonas;
}
```

Ejercicio 2

Parte 1)

1.a)

Primero supongamos que el arreglo no es modificado por el algoritmo.
Como todas las decisiones que toma el algoritmo solamente dependen de comparaciones de elementos del arreglo, y el arreglo no se modifica entonces las decisiones serian idénticas a que si la entrada fuese una permutación que tenga el mismo orden relativo. Al ser las mismas decisiones, las cantidad de comparaciones que se van a realizar es la misma y por ende el costo.

Si además se permite intercambios de elementos del arreglo. El orden relativo se sigue correspondiendo entre el arreglo y la permutación por lo que se siguen cumpliendo las propiedades anteriormente dichas.

Ejemplo: Si se tiene el arreglo *a=(20,10,23,1456,-2)* , la permutación $\pi =< 2,1,3,4,0 >$ tiene el mismo orden relativo.

1.b)
La cantidad de permutaciones de *n* elementos es finita. Al ser finita hace práctica la fórmula de caso medio vista en el Teórico.
Por ejemplo, si todas las permutaciones son equiprobables la probabilidad de que sea una permutación es *1/(n!)*.

Por otro lado, si consideramos todos los arreglos de *n* enteros equiprobables, la probabilidad de que sea un arreglo concreto es *0.* No pudiéndose utilizar la expresión de caso medio en su modalidad discreta, que es la que se ve en el curso.

1.c)
Definición: Sea *(p,q)* tal que $0 \leq p < q \leq n-1$ decimos que *(p,q)* es una inversión de π si *q* aparece antes que *p*.

Ejemplo: Dada la permutación $\pi =< 1,0,3,2,5,4,6,7 >$ tiene como inversiones (0,1), (2,3), (4,5).

2) Un ejemplo es $g(n) = 0$. $g \in O(1)$ ya que cumple la definición con $n_0 = 0$ y *c=1*. Por otro lado, $g \notin \Omega(1)$ ya que si $g \in \Omega(1)$ entonces existen un n_0 y *c>0* tal que $g(n) \geq c > 0$ para todo $n \geq n_0$. Luego fue absurdo suponer que $g \in \Omega(1)$. Se concluye que $g \notin \Omega(1)$ y finalmente $g \notin \Theta(1) = O(1) \cap \Omega(1)$.

3) Regla del límite.

Dadas dos funciones arbitrarias f: $N \rightarrow R^*$ y g: $N \rightarrow R^*$

- Si lím {n→+∞} (f(n) / g(n)) ∈ R^+ entonces f(n) ∈ O(g(n)) y g(n) ∈ O(f(n))
- Si lím {n→+∞} (f(n) / g(n)) = 0 entonces f(n) ∈ O(g(n)) pero g(n) ∉ O(f(n))
- Si lím {n→+∞} (f(n) / g(n)) = +∞ entonces f(n) ∉ O(g(n)) pero g(n) ∈ O(f(n))

Si $f(n) = 2 + (-1)^n$ y *g(n) = 1* se cumple que f(n) ∈ O(g(n)) y g(n) ∈ O(f(n)) pero el límite no existe.

Parte 2)

Dominio de definición:
Para este problema en particular $D_n = \{(a_0, a_1, .., a_{n-1}) / a_i \in Z, i \in \{0, ..., n-1\}\}$.

Análisis del costo
Se observa que las líneas que se tienen que contar para el costo de una entrada $e \in D_n$ son:

`swap(a, i, i+2)` tiene costo $2c_2$ y se ejecuta siempre $\left\lfloor \frac{n}{3} \right\rfloor$ veces.

`if(a[i]>a[j])` tiene costo c_1 y la cantidad de veces que se ejecuta es

`swap(a, i, j)` tiene costo $2c_2$ y la cantidad de veces que se ejecuta es variable.

Luego del primer for se realiza una permutación fija (que no depende del contenido del arreglo).

Sea un arreglo $a = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, ..., a_{n-4}, a_{n-3}, a_{n-2}, a_{n-1})$

La permutación de la primera iteración modifica bloques de largo tres. En caso que el tamaño del arreglo no sea múltiplo de tres no se intercambian el último o los dos últimos elementos.

Caso $n \equiv 0 (\mathrm{mod}\, 3)$:
Queda transformado en: $a' = (a_2, a_1, a_0, a_5, a_4, a_3, a_8, ..., a_{n-1}, a_{n-2}, a_{n-3})$

Caso $n \equiv 1 (\mathrm{mod}\, 3)$:
Queda transformado en: $a' = (a_2, a_1, a_0, ..., a_{n-7}, a_{n-2}, a_{n-3}, a_{n-4}, a_{n-1})$

Caso $n \equiv 2 (\mathrm{mod}\, 3)$:
Queda transformado en: $a' = (a_2, a_1, a_0, ..., a, a_{n-3}, a_{n-2}, a_{n-1})$

Para el mejor caso es condición suficiente y necesaria que el arreglo a' este ordenado en forma ascendente.
En ***términos del arreglo original*** las condiciones son:
i) $a_{i-1} > a_i > a_{i+1} \forall i \equiv 1 \bmod 3 \;\; 1 \le i < n-2$ condición de orden en cada subbloque de 3 elementos
ii) $a_i < a_{i+5} \forall i \equiv 0 \bmod 3 \;\; 0 \le i < n-5$ condición de orden entre subbloques (compara el que será el mayor de un bloque con el que será el menor del siguiente bloque tras el primer for)
nota: los subíndices refieren al arreglo original

Costo:

$$T_B(n) = 2c_2\left\lfloor \frac{n}{3} \right\rfloor + (c_1)\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = 2c_2\left\lfloor \frac{n}{3} \right\rfloor + (c_1)\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1$$

Para el peor caso es condición suficiente y necesaria que el arreglo a' este ordenado en forma descendente.
En ***términos del arreglo original*** las condiciones son:
i) $a_{i-1} < a_i < a_{i+1} \forall i \equiv 1 \bmod 3 \;\; 1 \le i < n-2$ condición de orden en cada subbloque de 3 elementos
ii) $a_i > a_{i+5} \forall i \equiv 0 \bmod 3 \;\; 0 \le i < n-5$ condición de orden entre subbloques (compara el que será el menor de un bloque con el que será el mayor del siguiente bloque tras el primer for)
nota: los subíndices refieren al arreglo original

Costo:

$$T_W(n) = 2c_2\left\lfloor \frac{n}{3} \right\rfloor + (c_1 + 2c_2)\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1$$

Operando

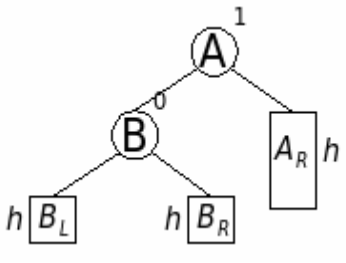
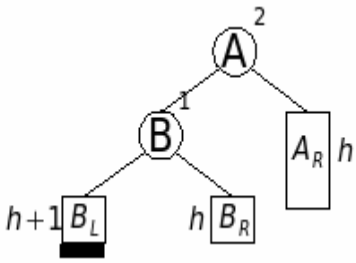
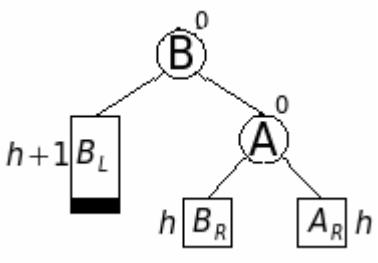
$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-1-(i+1)+1) = \sum_{i=0}^{n-1}(n-i-1)$$

$$= \sum_{i=0}^{n-1}(n-1) - \sum_{i=0}^{n-1} i = n(n-1) - \frac{(n-1)n}{2} = \frac{(n-1)n}{2}$$

Entonces

$$T_B(n) = 2c_2\left\lfloor \frac{n}{3} \right\rfloor + c_1\frac{(n-1)n}{2} \text{ y } T_W(n) = 2c_2\left\lfloor \frac{n}{3} \right\rfloor + (c_1 + 2c_2)\frac{(n-1)n}{2}$$

Ejercicio 3) Rotación Tipo LL

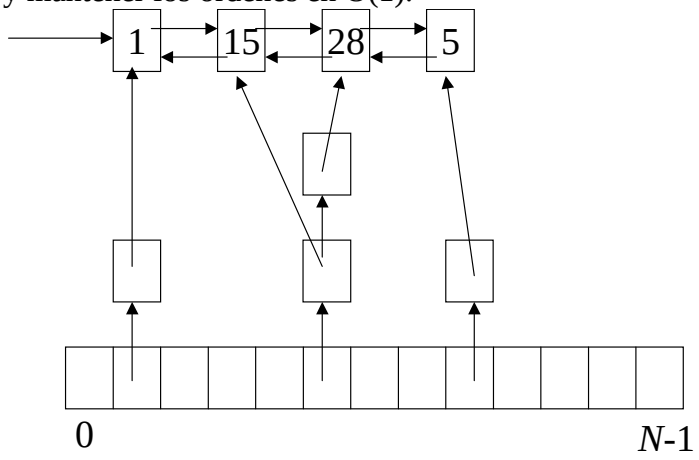
		
Previo a la inserción	**Luego de la inserción**	**Rebalanceo**

Ejercicio 4)

a) Asumiendo inicialización de los elementos en la creación.

Operación	Tiempo Promedio	Tiempo en el Peor Caso
`Crear`	O(*N*)	O(*N*)
`Insertar`	O(1)	O(1)
`Pertenece`	O(1)	O(*N*)
`Borrar`	O(1)	O(*N*)

b) Se agrega una lista doblemente enlazada que mantenga las claves ingresadas, al agregarse un elemento se agrega a la lista en cuestión. Debe ser doblemente enlazada para permitir la eliminación de claves y mantener los órdenes en O(1).



Operación	Modificación
`Crear`	Se crea la lista doblemente enlazada vacía.
`Insertar`	Se agrega el elemento al principio de la lista doblemente enlazada. Además, se referencia desde la tabla de hash.
`Pertenece`	Para comprobar si un elemento existe hay que desreferenciar el puntero que se encuentra en la lista del bucket de la tabla de hash. Este bucket es obtenido utilizando la función de hash.
`Borrar`	Al borrar un elemento se lo debe eliminar de la lista doblemente enlazada.

c) Se crea la estructura agregándole un AVL y al insertar cada elemento además de lo realizado en la parte anteriror, se inserta en un AVL con *orden total* de inserción de los M elementos igual a NLog(M), el retorno de los elementos de la lista se realiza recorriendo el AVL (log M).