Universidad Simón Bolívar
Teoría de Algoritmos
Tarea I

Fabiola Di Bartolo
Carnet: 09-87324

21 de octubre de 2009

1. Cormen - Ejercicio 17.3.6. Muestre cómo implementar una cola con dos pilas ordinarias tal que el costo amortizado de cada operación de `ENQUEUE` y `DEQUEUE` sea $O(1)$.

   Llamaremos S_1 y S_2 a las pilas que utilizaremos en la implementación de la cola. La pila S_1 la usaremos para encolar y la S_2 para desencolar. Para desencolar un elemento, la pila S_2 debe tener al menos uno. Esto significa, que previamente se deben haber movido los elementos de la pila S_1 a la pila S_2 y este proceso se realizará cuando se hayan extraido todos los elementos de la pila S_2 y se desee desencolar otro mas. Esto puede ser costoso para una iteración i, pero termina siendo beneficioso, lo cual se verá luego con el análisis del costo amortizado.

```
proc ENQUEUE(x)
  S1.push(x);
.
```

```
proc DEQUEUE(x)
  if S2.isEmpty()
    then
        while ¬S1.isEmpty() do
            x := S1.pop();
            S2.push(x);
        od
  fi;
  y := S2.pop();
  return y;
.
```

   Donde `push(x),pop(),isEmpty()` son los métodos conocidos de manejo de pilas.

Análisis del costo amortizado

Conociendo que:

- ⋆ Y la función potencial es:
  $\phi(S) = N\acute{u}mero\ de\ elementos\ de\ la\ pila\ S_1$
- ⋆ El costo amortizado es:
  $\hat{C}_i = C_i + \phi(S_i) - \phi(S_{i-1})$

Se procede al análisis:

- ⋆ Inicialmente no se tiene ningún elemento encolado, por lo tanto: $\phi(S_0) = 0$.
- ⋆ Se cumple para cualquier iteración i: $\phi(S_i) \geq \phi(S_0)$. Esto se debe a que la cantidad de elementos en la pila siempre será positiva o cero, porque la pila nunca va a deber elementos.
- ⋆ El costo amortizado para el procedimiento `ENQUEUE(x)` en una iteración i es: $\hat{C}_i = 1+(n+1)-(n) = 2$, donde el primer término se refiere al costo de ejecutar la instrucción de enpilar, el segundo indica la cantidad de elementos que tendrá la pila S_1 como resultado de la inserción, y el tercero indica la cantidad de elementos que tenía la pila S_1 para la iteración $i-1$, es decir, antes de insertar el nuevo elemento.
- ⋆ El costo amortizado para el procedimiento `DEQUEUE(x)` en una iteración i depende de si la pila S_2 se encuentra vacía o no:
  - ▷ Si la pila S_2 está llena, entonces la cantidad de los elementos en S_1 no cambia, es por esto que $\hat{C}_i = 1 + (n) - (n) = 1$.
  - ▷ Si la pila S_2 está vacía, se tiene que pagar el costo de mover todos los elementos (un por uno) de la pila S_1 a la pila S_2 para luego desencolar el último. entonces la cantidad de los elementos en S_1 no cambia, es por esto que $\hat{C}_i = (n+1) + (0) - (n) = 1$.

Por lo tanto, el costo amortizado de cada operación de estas es $O(1)$, lo que implica que el costo de una secuencia de n operaciones es $O(\mathrm{n})$.

2. Mostrar que la altura de un árbol AVL con n nodos es $O(\log n)$. (Ayuda: use inducción sobre n, ó establezca una recurrencia para Th = número mínimo de nodos de un árbol AVL de altura h, resuélvala y utilícela). Un árbol AVL es un árbol binario de búsqueda donde para cada nodo la diferencia de las alturas de sus dos subárboles es a lo sumo 1. Considere que el árbol vacío posee altura -1.

   Suposiciones para realizar la demostración:

   - ⋆ Un árbol AVL tiene la siguiente propiedad: $\mid h_{izq} - h_{der} \mid \leq 1$, donde h_{izq} y h_{der} son las alturas del subárbol izquierdo y derecho respectivamente del árbol AVL.
   - ⋆ Tomamos Th = número mínimo de nodos de un árbol AVL de altura h

   Demostración:

   - ⋆ Caso base 1: Suponiendo que altura de un árbol AVL que sólo contenga un nodo raíz es 0, tenemos que $T_0 = 1$
   - ⋆ Caso base 2: Para un árbol AVL de tamaño 2, tenemos que $T_1 = 2$
   - ⋆ Para $n \geq 2$: Sea A un árbol AVL de altura h y que tiene la mínima cantidad de nodos permitida, y sean A_{izq} y A_{der} sus subárboles izquierdo y derecho respectivamente, se sabe que por ser subárboles de A, ambos son AVL y tienen la mínima cantidad de nodos posible. Además por tener la mínima cantidad de nodos, uno de los dos subárboles hijos deben tener de altura $h - 2$, si se supone que el árbol A_{der} es el que la tiene, entonces el árbol A_{izq} tiene altura $h - 1$. De este razonamiento se puede deducir la siguiente recurrencia:

     $$T_h = T_{h-1} + T_{h-2} + 1 \tag{1}$$

     Donde, T_{h-1} representa el número de nodos del árbol A_{izq} y T_{h-2} representa el número de nodos del árbol A_{der}. La recurrencia anterior es similar a una recurrencia muy conocida, la recurrencia

utilizada para hallar el n-ésimo termino de Fibonacci que se utilizará en la demostración:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}. \text{ (Si } n \geq 2) \tag{2}$$

Esta recurrencia es homogenea, por lo que es más fácil de resolver. Además se sabe que A tiene al menos F_n nodos (para $n \geq 2$) , haciendo el cambio de variable de n por h en la ecuación anterior, queda lo siguiente:

$$n \geq T_h > F_h \tag{3}$$

Se le halla el polinomio característico a F_h:

$$x^2 - x - 1 = 0 \tag{4}$$

Resolviendo el polinomio carcterístico, quedan las raices: $\phi = \frac{1+\sqrt{5}}{2}$ y $\hat{\phi} = \frac{1-\sqrt{5}}{2}$, por lo que la formula cerrada de Fibonacci es de la siguiente forma:

$$F_h = \frac{\phi^h - \hat{\phi}^h}{\sqrt{5}} \approx \frac{\phi^h}{\sqrt{5}} \tag{5}$$

Sustituyendo F_h por la fórmula aproximada, queda:

$$n \geq T_h > \frac{\phi^h}{\sqrt{5}} \Rightarrow \log n > h\log\phi - \log\sqrt{5} \tag{6}$$

Tomando logaritmo en ambos lados

$$n \geq T_h > \frac{\phi^h}{\sqrt{5}} \Rightarrow \log n > h\log\phi - \log\sqrt{5} \tag{7}$$

$$\Rightarrow h < \frac{\log n + log\sqrt{5}}{\log\phi} \approx 1,44\log n + 1,67 \tag{8}$$

Con esto, se llega a la conclusión que la altura h de un árbol AVL para $h \geq 2$ es $O(logn)$.

3. *a)* Dar una instancia sencilla (4 objetos y números enteros) del problema de la mochila 0-1 donde al aplicar el algoritmo Greedy considerando los elementos por orden no creciente, o no decreciente, de peso; por orden no creciente, o no decreciente, de beneficio; ó por orden no creciente, o no decreciente, de beneficio por peso unitario (vi/wi), ninguno produce la solución óptima.

Si el algoritmo de *Greedy* recibe de entrada la siguiente instancia en orden ascendente o descendente para cualquiera de los 3 parámetros, no encontrará la solución óptima al problema $max \sum_{i=1}^{n} v_i x_i$ y $\sum_{i=1}^{n} w_i x_i \leq W$ para $x_i \in 0, 1$, donde v es el beneficio, w el peso del objeto y W la capacidad máxima de la mochila.

Instancia $(n = 4$**,**$W = 55)$

w	5	20	30	50
v	15	40	17	60
v/w	3	2	0,75	1,2

Análisis

- ⋆ Si la entrada se recibe ordenada por el parámetro w de forma ascendente, devuelve como solución el conjunto de elementos 1,2,3, donde $w = 5 + 20 + 30 = 55$ y $v = 15 + 40 + 17 = 72$.
- ⋆ Si la entrada se recibe ordenada por el parámetro w de forma descendente, devuelve como solución el conjunto de elementos 1, donde $w = 50$ y $v = 60$.
- ⋆ Si la entrada se recibe ordenada por el parámetro v de forma ascendente, devuelve como solución el conjunto de elementos 1,2,3, donde $w = 5 + 30 + 20 = 55$ y $v = 15 + 17 + 40 = 72$.

- ⋆ Si la entrada se recibe ordenada por el parámetro v de forma descendente, devuelve como solución el conjunto de elementos 1, donde $w = 50$ y $v = 60$.
- ⋆ Si la entrada se recibe ordenada por el parámetro v/w de forma ascendente, devuelve como solución el conjunto de elementos 3, donde $w = 30$ y $v = 17$.
- ⋆ Si la entrada se recibe ordenada por el parámetro v/w de forma descendente, devuelve como solución el conjunto de elementos 1,2, donde $w = 5 + 20 = 25$ y $v = 15 + 40 = 55$.
- ⋆ La solución óptima viene dada por el conjunto de elementos 1,4, donde $w = 50 + 5 = 55$ y $v = 60 + 15 = 75$, la cual no fue encontrada por el algoritmo.

b) Brassard - Ejercicio 6.18. En la sección 6.5 se suponía que estaban disponibles n objetos, numerados de 1 a n. Supongamos en cambio que tenemos n tipos de objetos disponibles, con un suministro adecuado de objetos de cada tipo. Formalmente, esto sólo cambia la restricción anterior $0 \leq x_i \leq 1$ por la restricción mas permisiva $x_i \geq 0$. ¿Sigue funcionando el algoritmo de *Greedy* de la sección 6.5?.¿Sigue siendo necesario?

Esta demostrado que para $0 \leq x_i \leq 1$ y recibiendo la entrada en orden decreciente por el parámetro v/w el algoritmo *Greedy* encuentra la solución óptima. Para este caso, el algoritmo *Greedy* devolverá únicamente el primer elemento de la lista ordenada, y esto es, porque no tiene ningun motivo para pasar al siguiente objeto, ya que no pasará al siguiente elemento hasta haber agotado el actual y como no se tiene ninguna restricción sobre la cantidad del objeto, lo único que lo limita es el peso W que la mochila puede aguantar.

Veamos si quedándose con el primer elemento, siempre encontrará la solución óptima. Para ello se compararán dos soluciones una encontrada por el algoritmo *Greedy* y otra por cualquier estrategia, que llamaremos B.

La solución del Greedy escoge únicamente el primer elemento de

la lista ordenada, es decir que $x_i = 0$ para $i \geq 2$, así que es de la forma:

$$W = w_1 x_1 \Rightarrow x_i = \frac{W}{w_1};\ V = v_1 x_1 \Rightarrow V = v_1 \frac{W}{w_1} \tag{9}$$

Suponga que V' es la solución encontrada por la estrategia B, así que la solución sería de la forma:

$$W' = \sum_{i=1}^{n} w_i y_i;\ V' = \sum_{i=1}^{n} v_i y_i. \tag{10}$$

Por definición del problema, se sabe también que $x_i \geq 0$, $y_i \geq 0$ y que $\forall i \frac{v_1}{w1} \geq \frac{v_i}{wi}$ (ya que asumimos que la entrada siempre va a estar ordenada de forma decreciente por el parámetro v/w). Se partirá suponiendo que V es mejor que V':

$$V - V' = \frac{W v_1}{w_1} - \sum_{i=1}^{n} v_i y_i = \frac{W v_1}{w_1} - \sum_{i=1}^{n} \frac{v_i}{w_i} y_i w_i \tag{11}$$

Como se dijo anteriormente: $\forall i \frac{v_1}{w1} \geq \frac{v_i}{wi}$, así que al sustiuir $\frac{v_i}{wi}$, queda lo siguiente:

$$V - V' \geq \frac{W v_1}{w_1} - \sum_{i=1}^{n} \frac{v_1}{w_1} y_i w_i \tag{12}$$

$$\Rightarrow \frac{W v_1}{w_1} - \frac{v_1}{w_1} \sum_{i=1}^{n} y_i w_i = \frac{v_1}{w_1}(W - W') \tag{13}$$

Debido a que el algoritmo *Greedy* llenará la mochila completamente, sabemos que $W - W' \geq 0$ es cierto. Con lo cual, queda demostrado que el algoritmo *Greedy* para este problema siempre encontrará una solución óptima. Sin embargo, no es necesario utilizarlo, con sólo ordenar la lista de elementos de forma decreciente por el parámetro v/w y escoger el primer elemento es suficiente para encontrar la solución óptima.

4. Dado un Grafo no dirigido simple (sin aristas múltiples) $G = (V, E)$, con E no vacío, la matroide de cociclos $M = (E, F)$ de G tiene como conjunto base a E y un independiente es un conjunto de aristas que al eliminarlas simultáneamente de G no aumenta el número de componentes conexas.

   a) Muestre que $M = (E, F)$ es una matroide.

   Para demostrar que M es una matroide, se tienen que cumplir las siguientes condiciones:

   - $\star$ $E \neq \emptyset$ y $F \neq \emptyset$.
     - $\triangleright$ La definición del problema dice que E no es vacío, es decir $E \neq \emptyset$.
     - $\triangleright$ F es la familia de subconjuntos de E que son independientes, por lo tanto, $\emptyset \subseteq E$ y $\emptyset$ es un conjunto independiente, ya que al no eliminar ninguna arista, no aumenta el número de componentes conexas, permanece el grafo intacto, por lo tanto $\emptyset \in F$, es decir que F tiene al menos un elemento, $F \neq \emptyset$.
   - $\star$ $\forall S' \in F[S'' \subseteq S' \Rightarrow S'' \in F]$
     - $\triangleright$ Si $S'' = \emptyset$, ya se se demostró que $\emptyset \in F$
     - $\triangleright$ Si $S'' = S$, ya se sabe que $S' \in F$, así que $S'' \in F$
     - $\triangleright$ Si $S'' \subset S$, es lo mismo que decir $[\forall x : x \in S'' \Rightarrow x \in S']$, y como S' es un independiente, significa que el subgrafo de G, $G' = (V, E - S')$ tiene la misma cantidad de componentes conexas que G, asi que si se le agrega a ese subgrafo las aristas de S' que no pertenecen a S'', es decir los $x \in S' - S''$, formando el subgrafo $G'' = (V, (E - S') \cup (S' - S''))$, evidentemente no aumenta el número de componentes conexas porque no se están eliminando nuevas aristas, más bien, se está eliminando sólo una parte de las aristas que ya se sabían que no generaban nuevas componentes conexas.
   - $\star$ $A \in F, B \in F$ y $\mid A \mid < \mid B \mid$ entonces $\exists x \in B - A$ tal que $A \cup \{x\} \in F$

- ▷ Sean $A = \emptyset$, $C = \{x\}$ y $C \subseteq B$, es evidente que se cumple, ya que $B - \emptyset = B'$, $\emptyset \cup \{x\} = C$ y con la regla anterior ya se había demostrado que cualquier subconjunto de un independiente, es un independiente, $C \in F$.
- ▷ Sea $A \neq \emptyset$, se tiene que $\mid B \mid \geq \mid B \mid +1$, y un conjunto es independiente si sus elementos están en ciclos distintos, no puede tener dos aristas que correspondan al mismo ciclo, ya que si se eliminan juntas, se crea una nueva componente conexa. Como en B existe al menos una arista más que en A, si esa arista está en una componente conexa distinta a las que están las aristas de A, entonces al agregarla al conjunto A, $A \cup \{x\}$, también será indepediente. Sin embargo, puede darse el caso que todas las aristas entre A y B estén en componentes conexas comunes, en ese caso, como A y B no tienen la misma cardinalidad, existe un elemento x en B que no está dentro los mismos ciclos en que se encuentran las aristas de A y ese al unirlo con A, $A \cup \{x\}$, no romperá la propiedad de independencia. Por lo tanto existe un x en $B - A$, tal que $A \cup \{x\} \in F$

Por lo tanto $M = (E, F)$ es una matroide.

b) Un subconjunto de E que no esté en F se dice que es dependiente. Un conjunto dependiente minimal es uno donde todo subconjunto propio es independiente. Caracterice en términos de grafos un dependiente minimal. Cómo serían los dependientes minimales de un ciclo elemental con n-vértices.

- ⋆ Sea D un conjunto dependiente, $D \subset E$, $D \notin F$. D es minimal si, $\mid D \mid = 2$ y las aristas pertencientes a D están dentro de un mismo ciclo, por lo tanto $\{x, y\} \notin F$ porque crearía una componente más al eliminar estas aristas, pero si $\{x\} \in F$ y $\{y\} \in F$.

- ⋆ Los dependientes minimales de un ciclo elemental con n-vertices serían los conjuntos formados por todas las combinaciones n en 2 de las aristas que conforman el ciclo, es decir que el número total de conjuntos dependientes minimales viene dado por la combinatoria $\binom{n}{2}$.

5. Brassard - Ejercicio 7.38. Tiene usted que organizar un torneo con n participantes. Cada participante tiene que competir exactamente una vez con todos los posibles oponentes. Además, cada participante tiene que jugar exactamente un partido cada día, con la posible excepción de un solo día en el cual no juega.

   a) Si n es potencia de 2, dar un algoritmo para construir un horario que permita que el torneo concluya en $n - 1$ días.

   Para resolver este problema con n par, se escribirá n como $n = 2k$ donde k es un entero positivo. Se tienen los siguientes casos:

   - ⋆ Para el caso base $n = 2$, se soluciona haciendolos jugar a los únicos dos jugadores, ese día 1.
   - ⋆ Para $n \geq 2$, se subdivide el conjunto de participantes en dos subconjuntos disjuntos, el conjunto P_1 formado por los participantes de 1 a $k - 1$ y el conjunto P_2 otro formado por los participantes de k a n. Estos subconjuntos se vuelven a subdividir en dos recursivamente hasta llegar a conjuntos de tamaño 2 (caso base) y después de conseguir las soluciones a estos dos subproblemas, se deben combinar ambas soluciones y se combinan hasta llegar al primer nivel de la recursión. La combinación se hace debido a que faltan los juegos de los participantes pertencientes P_1, contra los de P_2 y viceversa. Por ejemplo, para completar los juegos del primer participante, lo que se hace es asignarle las competencias los días sucesivos con los participantes de P_2 en orden creciente $(k..n)$.

   La salida de este algoritmo *Divide and Conquer* será una matriz $m(nXn-1)$ donde la casilla $m_{i,j}$ contendrá el número del participante contra el cual el participante i-ésimo competirá en el día j-ésimo.

Algorimo para crear el horario para n par

```
/* m es la matriz que representa al horario */
/* Se llama inicialmente a crearHorario(m,1,n) */
proc crearHorario(m, ini, fin)
  if ini − fin == 1
    then
        Caso base, compiten entre ambos ese dia
        m[ini, 1] := fin;
        m[fin, 1] := ini;
     else
        mitad := (ini + fin)div2;
        /* Subsoluciones */
        crearHorario(m, ini, mitad);
        crearHorario(m, mitad + 1, fin);
        /* Completa el calendario */
        completarHorario(m, ini, mitad, mitad, fin − 1, mitad + 1);
        completarHorario(m, mitad + 1, fin, mitad, fin − 1, ini);
  fi
.

proc completarHorario(m, pInf, pSup, diaInf, diaSup, pPartida)
  for j := diaInf to diaSup do
     m[pInf, j] := pPartida + j + diaInf;
  od
  for i := pInf + 1 to pSup do
      /* Ultimo contrincante de i-1 es primer contricante de i */
     m[i, diaInf] := m[i − 1, diaSup];
  od
  for j := diaInf + 1 to diaSup do
     /* Contrincante de i-1 es el contrincante de hoy para i */
     m[i, j] := m[i − 1, j − 1];
  od
.
```

b) Para todo entero $n > 1$, dar un algoritmo para construir un horario que permita que el torneo concluya en $n-1$ días si n es par, o n días si n es impar.

Este algoritmo debe considerar los siguientes casos:

- ⋆ Si n es par, el proceso es similar al anterior.
- ⋆ Si n impar, se incrementa en uno n, es decir suponemos que tenemos un jugador ficticio, de esta manera queda n par y el algoritmo hallará una solución. Para implementar esto, se realizarían las siguientes modificaciones:
  - ▷ Se tiene una variable global `nPar` que tiene el valor `true` si la variable `n` de entrada es par o `false` si es impar.
  - ▷ Se realiza la asignación `n:=n+1` en la inicialización del algoritmo.
  - ▷ Cuando el algoritmo este escribiendo o accediendo el elemento $m_{i,j}$ donde `i` corresponda al número del jugador ficticio (`n`) y el valor de `nPar` sea igual a `false`, se considerará como único valor posible del elemento $m_{i,j}$ blanco (`' '`).

6. Brassard - Ejercicio 7.40. Considere una matriz $T[1..n]$ y un entero k entre 1 y n. Utilice una simplificación para diseñar un algoritmo eficiente que intercambie los k primeros elementos de T con los $n-k$ últimos, sin hacer uso de una matriz auxiliar. Analice el tiempo de ejecución de su algoritmo.

Para el intercambio de los primeros k elementos con los últimos $n-k$ elementos, se tienen varios casos, asumimos que k puede tener un valor entre 0 y n:

- ⋆ Caso 1: Si $k = n$ ó $k = 0$, no se realiza ningún intercambio, la entrada al algoritmo es la salida del mismo.

- ⋆ Caso 2: Si $k = n - k$, significa que $n = 2k$, para este caso se toma uno a uno cada elemento perteneciente a los k primeros y se van intercambiando con los últimos.
- ⋆ Caso 3: Si $k < n - k$ ó si $k > n - k$, no se puede realizar el intercambio completo, ya que existen elementos que no tienen su elemento correspondiente para ser intercambiado. Es por esto, que tomamos m que será el mínimo entre k y $n - k$ y se realizará el intercambio de igual manera que para el caso 2. Es importante observar que $m = Min(k, n - k) \leq ndiv2$, decimos $ndiv2$ y no $n/2$ porque en este caso n puede ser impar.

Algoritmo de intercambio

```
begin
  /* Entrada: M, k, n */
  /* Salida: M */
  if k == n ∨ k == 0
    then exit;
    else
        m := Min(k, n − k)
        for i := 0 to m − 1 do
            x := M[i];
            y := M[n − i];
            M[i] = y;
            M[n − i] = x;
        od
  fi
end
```

El orden de este algoritmo esta directamente relacionado con la cantidad de elementos a intercambiar, y sabemos que el peor caso es que $k = n - k$, es decir que se realicen $n/2$ iteraciones. Por lo tanto el tiempo de ejecución de el algoritmo $T(n)$ es $O(n/2)$