Algoritmos y Estructuras de Datos II - 3 de julio de 2013

Examen Final Teórico-Práctico

Docentes: Daniel Fridlender, Silvia Pelozo y Alejandro Tiraboschi

Alumno: .... Email: ....

Incluir SIEMPRE justificaciones de sus respuestas con la mayor claridad posible.

1. Se define el TAD silueta, que representa la silueta que dejan los edificios de una ciudad en el horizonte.

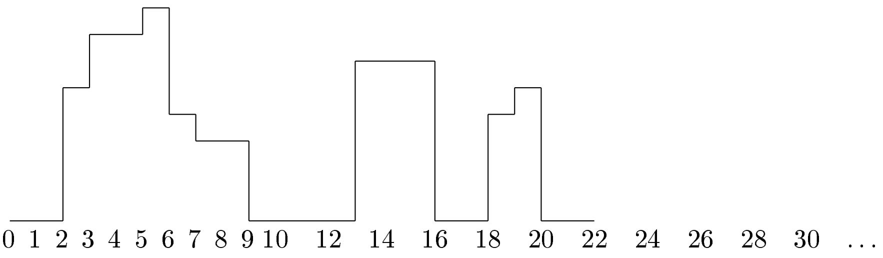


Figura 1:

El tipo abstracto tiene 2 constructores: uno para crear la silueta vacía, es decir, sin edificios y otro para agregar un edificio a la izquierda de una silueta preexistente. Este último constructor tiene por argumento dos naturales a y h y una silueta s y agrega a la izquierda de s un edificio de ancho a y altura h.

TAD silueta
constructores
vacía : silueta
ag_edif : natural × natural × silueta → silueta
operaciones
un_edif : natural × natural × natural → silueta
levantar : silueta → silueta
hundir : silueta → silueta
...
ecuaciones
ag_edif(a,h,ag_edif(b,h,s)) = ag_edif(a+b,h,s)
ag_edif(0,h,s) = s

un_edif(d,a,h) = ag_edif(d,0,ag_edif(a,h,vacia))

levantar(vacía) = vacía
levantar(ag_edif(a,h,s)) = ag_edif(a,h+1,levantar(s))

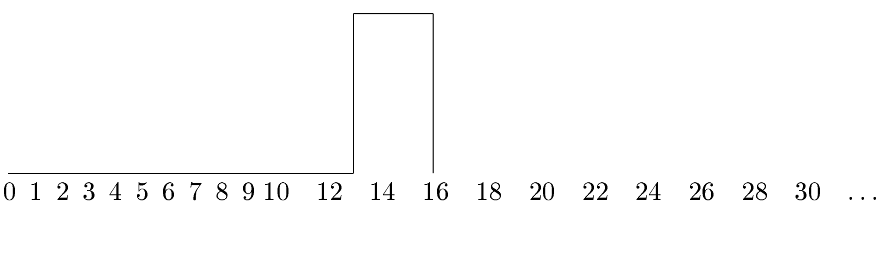
hundir(vacía) = vacía
hundir(ag_edif(a,0,s)) = ag_edif(a,0,hundir(s))
h > 0 ⟹ hundir(ag_edif(a,h,s)) = ag_edif(a,h-1,hundir(s))

Así, la silueta de la Figura 1 puede ser construida como
s = ag_edif(2,0,ag_edif(1,5,ag_edif(2,7,ag_edif(1,8,ag_edif(1,4,ag_edif(2,3,ag_edif(4,0,s’)))))))
donde s’ = ag_edif(3,6,ag_edif(2,0,ag_edif(1,4,ag_edif(1,5,ag_edif(2,0,vacía))))). La utilización de s’ se debe únicamente a que no hay suficiente espacio en una sola línea para toda la definición de s.

Una misma silueta puede construirse de diferentes maneras, por ejemplo
s’ = ag_edif(1,6,ag_edif(2,6,ag_edif(2,0,ag_edif(0,8,ag_edif(1,4,ag_edif(1,5,ag_edif(2,0,vacía))))))).

La primera ecuación del TAD dice justamente que dos edificios adyacentes de igual altura producen el mismo efecto que un solo edificio suficientemente ancho. Y la segunda ecuación dice que un edificio de ancho 0 no afecta la silueta. No así un edificio de altura 0 pero ancho positivo, ya que puede funcionar como separador de dos edificios.

La operación un_edif se usa para obtener la silueta de un edificio solo. Por ejemplo, un_edif(13,3,6) es la silueta



a) Implementar el TAD silueta con una lista enlazada según la siguiente definición. Se utiliza el nombre skyline para la representación de la silueta.

```
type node = tuple
                width: nat
                height: nat
                next: pointer to node
            end
type skyline = pointer to node
```

Con esta implementación, la silueta s de la Figura 1 se representaría de la siguiente manera:

s [ ] → [2|0] → [1|5] → [2|7] → [1|8] → [1|4] → [2|3] → [4|0] → [3|6] → [2|0] → [1|4] → [1|5] → [2|0] →

Ésta es la representación simplificada. Otra representación (no simplificada) es posible reemplazando el tercer nodo por dos nodos con campo width igual a 1 y campo height igual a 7, y otra agregando nodos con campo width igual a 0 y campo height arbitrario. **Implementar** un procedimiento **auxiliar simplify**, que aplicado a un skyline devuelva la representación simplificada: sin nodos con width = 0 ni nodos consecutivos de igual height. Este procedimiento debe modificar su argumento, y liberar los nodos que corresponda.

Implementar todas las operaciones del TAD silueta. Se asume que las operaciones que reciben una silueta la reciben simplificada y no la destruyen ni modifican, y las que devuelven siluetas, deben devolverlas también simplificadas, por ejemplo, valiéndose del procedimiento simplify.

b) Agregar a la especificación una operación que devuelva la altura máxima de la silueta, otra que devuelva la superficie bajo la silueta, y otra que permita agregar un edificio a la derecha de una silueta.

2. Dado un grafo (V, L) donde $V = \{1, \ldots, n\}$ es el conjunto de vértices y L es la matriz que asocia un costo no negativo a cada una de las aristas, el algoritmo de Dijkstra calcula el costo del camino de menor costo a cada uno de los destinos posibles partiendo desde un vértice $v \in V$:

```
fun Dijkstra(L: array[1..n,1..n] of nat, v: {1,...,n}) ret D: array[1..n] of nat
    var c: {1,...,n}
    C:= {1,2,...,n}
    for j:= 1 to n do D[j]:= ∞ od
    D[v]:= 0
    do n-1 times → c:= elemento de C que minimice D[c]
                   C:= C-{c}
                   for j in C do D[j]:= mín(D[j],D[c]+L[c,j]) od
    od
end fun
```

A continuación, se propone un problema relacionado: se supone que los vértices son paradas o puntos de enlace entre diferentes servicios de transporte. Desde un punto x a otro punto y puede haber 0, 1 ó varios servicios, por ejemplo, puede haber unos a la mañana y otros a la tarde, puede haber normales y diferenciales. Lo que importa es que entre cada par de vértices existe **una lista de servicios** y cada uno de esos servicios tiene **horario de partida y de llegada**. En efecto, entre los puntos x e y puede haber servicios de distinta duración.

Por ello, a diferencia del caso del algoritmo de Dijkstra, $L[x, y]$ es una lista (vacía en caso de no haber servicios de x a y) de servicios. Esta lista puede asumirse ordenada según el orden que a usted le convenga (en ese caso **explicar cuál es ese orden**).

Dar un algoritmo que, dado un vértice de origen $v \in V$, y asumiendo que uno se encuentra allí en el tiempo $t = 0$, calcule para cada destino el menor tiempo en que puede llegarse a ese destino usando los servicios de transporte disponibles. Puede convenirte asumir definidos los tipos time (tiempo) y schedule (horario) con sus campos departure (partida) y arrival (llegada):

```
type time = nat
type schedule = tuple
                    departure: time
                    arrival: time
                end tuple
```

Con este tipo, la función a implementar tendrá el siguiente encabezado

```
fun conecciones(L: array[1..n,1..n] of list of schedule, v: {1,...,n}) ret D: array[1..n] of time
```

3. El número combinatorio $\begin{pmatrix} n \\ k \end{pmatrix}$, para $n, k \in \mathbb{N}$ con $k \leq n$ puede definirse recursivamente de la siguiente manera:

$$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{cases} 1 & \text{si } k = 0 \vee k = n \\ \begin{pmatrix} n-1 \\ k-1 \end{pmatrix} + \begin{pmatrix} n-1 \\ k \end{pmatrix} & \text{si } 0 < k < n \end{cases}$$

Posiblemente conozcas esta otra definición

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!\ (n-k)!}$$

pero la que interesa en este ejercicio es la primera.

Escribir un algoritmo que utilice programación dinámica para calcular el número combinatorio $\begin{pmatrix} n \\ k \end{pmatrix}$ valiéndose de una tabla (una matriz m) definida localmente (dentro de la función). Como el número combinatorio solamente está definido para $k \leq n$ no todas las posiciones de la matriz m deben calcularse. Para que su solución sea correcta, es fundamental que las celdas que intervengan en el cómputo del valor $m[i, j]$ hayan sido calculados con anterioridad.

A modo de ejemplo, si la función es invocada con $n = 7$ y $k = 4$ calculará la siguiente tabla y devolverá 35.

k=	0	1	2	3	4
n=0	1				
n=1	1	1			
n=2	1	2	1		
n=3	1	3	3	1	
n=4	1	4	6	4	1
n=5	1	5	10	10	5
n=6	1	6	15	20	15
n=7	1	7	21	35	35

¿Qué puede decir en este caso sobre el orden del algoritmo recursivo y el orden del algoritmo que utiliza programación dinámica?

4. Calcular el orden de cada uno de los siguientes algoritmos

   a) **proc** P(**in** n:**nat**)
       for i:= 1 **to** n **do**
           for j:= i **to** i+500 **do**
               $\mathcal{O}(1)$
           od
       od
   end proc

   b) **proc** Q(**in** n:**nat**)
       var m : **nat**
       m:= n ÷ 2
       for i:= 1 **to** 3 **do**
           Q(m)
           P(n)
       od
   end proc

5. (para alumnos libres) Dar la forma general de los algoritmos divide y vencerás. Explicarla.