Algoritmos y Estructuras
de Datos II

EJERCICIOS RESUELTOS - PRÁCTICO 8 - ALGORITMOS VORACES

Pedro Villar

Ejercicio 1

Demostrar que el algoritmo voraz para el problema de la mochila sin fragmentación no siempre obtiene la solución óptima. Para ello puede modificar el algoritmo visto en clase de manera de que no permita fragmentación y encontrar un ejemplo para el cual no halla la solución óptima.

Ejercicio 2

Considere el problema de dar cambio. Pruebe o dé un contraejemplo: si el valor de cada moneda es al menos el doble de la anterior, y la moneda de menor valor es 1, entonces el algoritmo voraz arroja siempre una solución óptima.

Solución

Falso, supongamos que se quiere dar un cambio de 15 sentavos y se tienen monedas de 1,5,12. El algoritmo voraz tomaría 12, 1, 1, 1, en total 4 monedas, pero la solución óptima sería 5, 5, 5, en total 3 monedas.

Ejercicio 3

Se desea realizar un viaje en un automóvil con autonomía A (en kilómetros), desde la localidad l_0 hasta la localidad l_n pasando por las localidades $l_1, ..., l_{n-1}$ en ese orden. Se conoce cada distancia $d_i \leq A$ entre la localidad li_{-1} y la localidad li (para $1 \leq i \leq n$), y se sabe que existe una estación de combustible en cada una de las localidades.

Escribir un algoritmo que compute el menor número de veces que es necesario cargar combustible para realizar el viaje, y las localidades donde se realizaría la carga.

Suponer que inicialmente el tanque de combustible se encuentra vacío y que todas las estaciones de servicio cuentan con suficiente combustible.

Solución

```
fun min_cargas(A :Float, d: List of Float) ret r: Nat
  var i: Nat
  var d_aux : List of Float
  d_aux := copy_list(d)
  r := 0
  do not is_empty(d_aux) ->
    l := seleccion(d_aux, A)
    drop(d_aux, l)
    r := r + 1
  od
  destroy_list(d_aux)
end fun
fun seleccion(d: List of Float, A: Float) ret l: Nat
  var sum, i: Nat
  var e: Float
  sum := 0
  i := 0
  l := 0
  do sum <= A && i < length(d) ->
    e := sum + index(d, i)
    if e <= A then
      sum := e
      l := i
    fi
    if e > A then
      i := length(d)
    fi
    i := i + 1
  od
end fun
```

Ejercicio 4

En numerosas oportunidades se ha observado que cientos de ballenas nadan juntas hacia la costa y quedan varadas en la playa sin poder moverse. Algunos sostienen que se debe a una pérdida de orientación posiblemente causada por la contaminación sonora de los océanos que interferiría con su capacidad de inter-comunicación. En estos casos los equipos de rescate realizan enormes esfuerzos para regresarlas al interior del mar y salvar sus vidas.

Se encuentran n ballenas varadas en una playa y se conocen los tiempos $s_1, s_2, ..., s_n$ que cada ballena es capaz de sobrevivir hasta que la asista un equipo de rescate. Dar un algoritmo voraz que determine el orden en que deben ser rescatadas para salvar el mayor número posible de ellas, asumiendo que llevar una ballena mar adentro toma tiempo constante t, que hay un único equipo de rescate y que una ballena no muere mientras está siendo regresada mar adentro.

Solución

```
type Ballena = tuple
                 id: Nat
                 s: Float
               end tuple

fun rescate(ballenas : set of Ballena) ret res : Queue of Ballena
  var v : set of Ballena
  var salvada : Ballena
  var t : Float
  v := copy_set(ballenas)
  res := empty_queue()
  do not is_empty(v) ->
    salvada := selec_ballena(v)
    enqueue(res, salvada)
    set_elim(v, salvada)
    t := t + salvada.s
    v := quit_muertas(v, t)
  od
  destroy_set(v)
end fun

fun selec_ballena(v : set of Ballena) ret r : Ballena
  var bs : set of Ballena
  var b : Ballena
  r := set_get(bs)
  do not is_empty(bs) ->
    b := set_get(bs)
    set_elim(bs, b)
    if b.s < r.s then
      r := b
    fi
  od
  destroy_set(bs)
end fun
```