Solución Segundo Parcial de Programación 3 (27/11/2010)

Instituto de Computación, Facultad de Ingeniería

Ejercicio 1 (25 puntos)

a)

Forma de la solución:

Tupla T de largo fijo L de la forma $T = < (x_1, y_1),...,(x_L, y_L) >$, siendo L la cantidad de vueltas. Cada componente (x_i, y_i) donde $1 \leq i \leq L$, representa:

- x_i : índice de la velocidad a la que se realiza la vuelta *i.*
- y_i : energía que tiene el ciclista al comenzar la vuelta *i.*

Restricciones explicitas:

$x_i \in \{1,\ldots,S\} \quad \forall i\, 1 \leq i \leq L$ (Para todas las vueltas, el índice de la velocidad debe ser uno de los S posibles)

$0 < y_i \leq C \quad \forall i\, 1 \leq i \leq L$ (La energía del ciclista nunca puede ser mayor a su energía máxima C, y debe ser mayor a cero porque el ciclista no puede quedarse sin calorías en ningún momento)

$y_1 = C$ (La energía del ciclista al comenzar la carrera esta completa)

La cantidad de calorías que debe tener el ciclista al comenzar la vuelta *i* debe ser mayor a las calorías que gastará utilizando la velocidad de índice x_i.

Entonces: $C[x_i] < y_i \quad \forall i\, 1 \leq i \leq L$

Restricciones implícitas:

Al evaluar los posibles valores de la componente (x_i, y_i) de la tupla, el valor de y_i está determinado por una componente anterior en la tupla (la componente *i-1*):

debe ser $y_i = y_{i-1} - C[x_{i-1}]$ en caso de que el ciclista haya decidido no alimentarse y en otro caso, el ciclista se alimenta complementando la energía que le quedó al final de la vuelta i-1 hasta completar C calorías (entonces y_i = C).

$\forall i\, 2 \leq i \leq L \quad y_i = y_{i-1} - C[x_{i-1}]$ ó $y_i = y_{i-1} - C[x_{i-1}] + (C - (y_{i-1} - C[x_{i-1}]))$

Función objetivo:

Se utiliza una variable auxiliar z_i que indica si el ciclista decidió alimentarse o no al comenzar la vuelta. Si este decidió hacerlo, tendrá toda su energía recargada para comenzar la vuelta y $z_i = 1$. En otro caso $z_i = 0$.

Entonces: $z_i = 1$ si $y_i = C$, y $z_i = 0$ en otro caso.

La función objetivo será:

$\min(\sum_{i=2}^{L} (t_i + B.z_i) + t_1)$, donde t_i es el tiempo en segundos que requiere dar una vuelta a la velocidad *i.*

b)

I. No es predicado de poda ya que el enunciado se deriva de la función objetivo, y a estos no se los considera predicados de poda.
II. La condición puede ser utilizada como predicado de poda, ya que la tupla en construcción no podrá mejorar el tiempo de la mejor solución encontrada hasta el momento. Además, la tupla debe cumplir con las restricciones implícitas y el valor objetivo no debe ser mayor al valor objetivo de la mejor solución encontrada hasta el momento.
III. La condición puede ser utilizada como predicado de poda, dado que para la construcción de la tupla no consideramos ciertas velocidades, reduciendo así el espacio de soluciones. Además, la tupla debe cumplir con las restricciones implícitas y el valor objetivo no debe ser mayor al valor objetivo de la mejor solución encontrada hasta el momento.
IV. No es un predicado de poda ya que el enunciado se deriva de la restricción implícita, y a estos no se los considera predicados de poda

c)

I. Todos los enunciados de la parte anterior son funciones de poda en el algoritmo de backtracking. Ellos se pueden implementar de la siguiente manera:

   b-i) Se debería tener en variables auxiliares el tiempo de la mejor tupla encontrada hasta el momento y el tiempo que va acumulando la tupla en construcción. Para implementar este ítem basta con comparar dichos valores y descartar la tupla en construcción si su tiempo es mayor o igual al de la mejor solución hasta el momento.

   b-ii) Teniendo las variables planteadas en la parte anterior y otra variable auxiliar que represente la cantidad de vueltas que le falta dar al ciclista, Para implementar este ítem se podría realizar el siguiente calculo:
   si el tiempo que va acumulando la tupla en construcción, sumado a la cantidad de vueltas que le faltan dar al ciclista multiplicado por el tiempo requiere la mayor velocidad; es mayor al tiempo de la mejor solución hasta el momento, entonces descarto la tupla que estoy construyendo.

   b-iii) Para implementar este ítem, tenemos que tener en cuenta el vector que indica para cada velocidad, el tiempo que esta requiere para dar la vuelta; y también el tiempo de detención. Para elegir la velocidad con la que queremos realizar una vuelta, descartamos aquellas que no cumplan con las restricciones implícitas, y de las restantes, teniendo en cuenta el tiempo de la máxima velocidad (buscamos el menor valor del vector de tiempos) sumado al tiempo B, comparamos ese valor con los tiempos de las velocidades disponibles, y descartamos aquellas cuyo tiempo sea mayor.

   b-iv) Para implementar este ítem, podemos ver que cuando construimos la tupla, y queremos agregar a ella el i-ésimo componente, primero obtengo y_i, y luego

determino la velocidad x_i. Para elegir este valor, solo elijo aquellos x_i tales que $c_{x_i} < y_i \quad \forall\, i\, 1 \le i \le L$

II. El conjunto de velocidades que deja de considerarse es independiente de la vuelta. Podemos realizar una nueva implementación, eliminando las velocidades que no se usarán al comienzo (antes de implementar el algoritmo de backtracking) y luego ejecutar el algoritmo, solo con las velocidades que tienen oportunidad de ser utilizadas (aquellas cuyo tiempo sea menor o igual que la suma del tiempo que lleva dar la vuelta a la máxima velocidad más el tiempo de detención B)

Ejercicio 2 (15 puntos)

Sea ***f(l, k)*** la distancia que se debe recorrer todavía, cuando se han marcado ***l*** lugares y un integrante del equipo está en el lugar l_k y el otro integrante está en el lugar l_l, siendo **l > k.**

La función ***f(l, k)*** tiene solución trivial si ***l = n*** ya que en ese caso se han marcado todos los lugares y la distancia que falta por recorrer es 0.

En general, se tienen dos opciones:

- que el integrante que se encuentra en el lugar l_l vaya al lugar l_{l+1} y resolver el problema ***f(l + 1, k)***.
- que el integrante que se encuentra en en l_k vaya al lugar l_{l+1} y resolver el problema ***f*(l + 1, l)**.

Por tanto, función recursiva que resuelve el problema es:

$$f(l,k) = \begin{cases} 0 \quad si \quad l = n \\ \min\{dis\tan cia(l,l+1) + f(l+1,k), dis\tan cia(k,l+1) + f(l+1,l)\} \quad si \quad l < n \end{cases}$$

La invocación de la función, para resolver el problema es ***f(1,0)***

Ejercicio 3 (20 puntos)

a) (10 puntos)

i.

Versión del Teórico: Mergesort utiliza una operación MERGE para combinar las soluciones en el proceso D&C. Esta operación recibe dos secuencias ordenadas, las intercala ordenadamente devolviendo una única secuencia. Su especificación será:

Entrada: Secuencias *S1 y S2* a intercalar
Salida: Secuencia ordenada
MERGE: SECxSEC → SEC **MERGE(S1,S2)=** **Si Vacia(S1)⇒ S2** **Sino** **Si Vacia(S2)⇒ S1** **Sino** **Si Primero(S1)< Primero(S2)** **InsFront(Primero(S1), MERGE(Resto(S1),S2))** **Sino** **InsFront(Primero(S2), MERGE(S1, Resto(S2))**

Se realizará la ordenación sobre el mismo arreglo. Para el MergeSort se utilizarán los valores p y q para indicar el comienzo y fin de la subsecuencia a ordenar. Para el MERGE se deben indicar ambos índices para las dos subsecuencias.

Con estas consideraciones el cabezal del MERGE quedaría:

void MERGE(arreglo &A, int ini1, int fin1, int ini2, int fin2);

No se realizará la implementación de este algoritmo. Cabe aclarar que MERGE (y por lo tanto MergeSort) utiliza espacio extra transitorio para realizar la intercalación, devolviendo el resultado en el arreglo original A.

```
void MergeSort( arreglo &A, int p, int q){
int medio;

if (p<q) {              // quedan al menos 2 elementos
  medio = (p+q) / 2;
  MergeSort(A, p, medio);
  MergeSort(A, medio+1, q);
  MERGE(A, p, medio, medio+1, q);
  } //if
} // fin Mergesort
```

Versión Alternativa: Mergesort utiliza aritmética de punteros para representar las subsecuencias S1 y S2 en que se subdivide e invoca a la operación MERGE para combinar las soluciones.

```
void merge(int* arreglo, int m_1, int m_2) // S = arreglo, m_1:  largo de S1, m_2: largo de S2
                int i = 0;
                int t_1 = 0; // comienzo (relativo) de S1
                int t_2 = 0; // m_1+ t_2: comienzo (relativo) de S2

                int* aux = new int[m_1 + m_2]; // memoria auxiliar

                while (t_1 < m_1 && t_2 < m_2) { // mientras no se terminen S1 y S2
                    int x = arreglo[t_1]; //x: primero(S1)
                    int y = arreglo[m_1 + t_2]; // y: primero(S2)

                    if ( x < y ) { // comparación de elementos: los dos primeros
                            aux[i] = x;
                            t_1++;
                    }
                    else {
                            aux[i] = y;
                            t_2++;
                    }
                    i++;
                }

                while (t_1 < m_1) { // Se terminó S2, se copia el resto de S1 al auxiliar
                    aux[i] = arreglo[t_1];
                    t_1++;
                    i++;
                }

                while (t_2 < m_2) { // Se terminó S1, se copia el resto de S2 al auxiliar
                    aux[i] = arreglo[m_1 + t_2];
                    t_2++;
                    i++;
                }

                for(int j=0; j < m_1 + m_2; j++) { // Se copia nuevamente al original
                    arreglo[j] = aux[j];
                }

                delete[] aux;
}

void mergeSort(int* arreglo, int n) {
```

```
        if (n>1) {
            int m_1 = n/2;
            int m_2 = n/2 + n%2;

            mergeSort(arreglo, m_1); // arreglo: comienzo de S1
            mergeSort(arreglo + m_1, m_2); // arreglo-m_1: comienzo de S2

            merge(arreglo, m_1, m_2); // Merge con S, largoS1, largoS2
        }
}
```

ii.

MERGE: realiza n-1 comparaciones en su peor caso, siendo n = largo (S1) + largo(S2)

$$T_w(n) = T_w(\lfloor (n/2) \rfloor) + T_w(\lceil n/2 \rceil) + (n-1)$$

$$T_w(1) = 0$$

Considerando n potencia de 2, $\mathbf{n=2^p}$ se asegura que cada vez que se divida por 2 la división será exacta. Entonces la recurrencia queda:

$$T_w(n) = 2T_w(n/2) + n - 1 \qquad T(1) = 0$$

Expandiendo la recurrencia y multiplicando sucesivamente por 2:

$$T_w(n) = 2T_w(n/2) + n - 1$$
$$2^1 T_w(n/2) = 2^2 T_w(n/2^2) + n - 2^1$$
$$2^2 T_w(n/2^2) = 2^3 T_w(n/2^3) + n - 2^2$$
$$\vdots$$
$$2^{p-1} T_w(n/2^{p-1}) = 2^p T_w(n/2^p) + n - 2^{p-1}$$
$$2^p T_w(n/2^p) = 2^p T_w(1) = 0$$

Sumando y eliminando términos queda:

$$T(n) = \sum_{i=0}^{p-1} (n - 2^i) \qquad T(n) = np - \sum_{i=0}^{p-1} 2^i \qquad T(n) = np - 2^p + 1$$

como habíamos tomado $n=2^p$, $p = \log n$

$$T_w(n) = n\log n - n + 1$$

i. ¿Mergesort es óptimo en el peor caso? Justifique

En el teórico del curso se probó que para un problema de ordenación basado en comparaciones $F_w(n) \in \theta(n \log n)$ siendo este el mejor orden que pueda alcanzar un algoritmo de ordenación tomando como operación básica comparación. Por otro lado, en la parte **a- ii** se prueba que $T_w(n) = n \log n - n + 1$. Utilizando la regla del límite se prueba que $T_w(n) \in \theta(n \log n)$ concluyéndose que es óptimo.

b) (10 puntos)

i.

- Un Árbol de Decisión (AD) es un árbol binario estricto (cada nodo tiene 2 hijos o ninguno) donde los nodos internos tienen 2 hijos y los externos (hojas) no tienen ninguno.

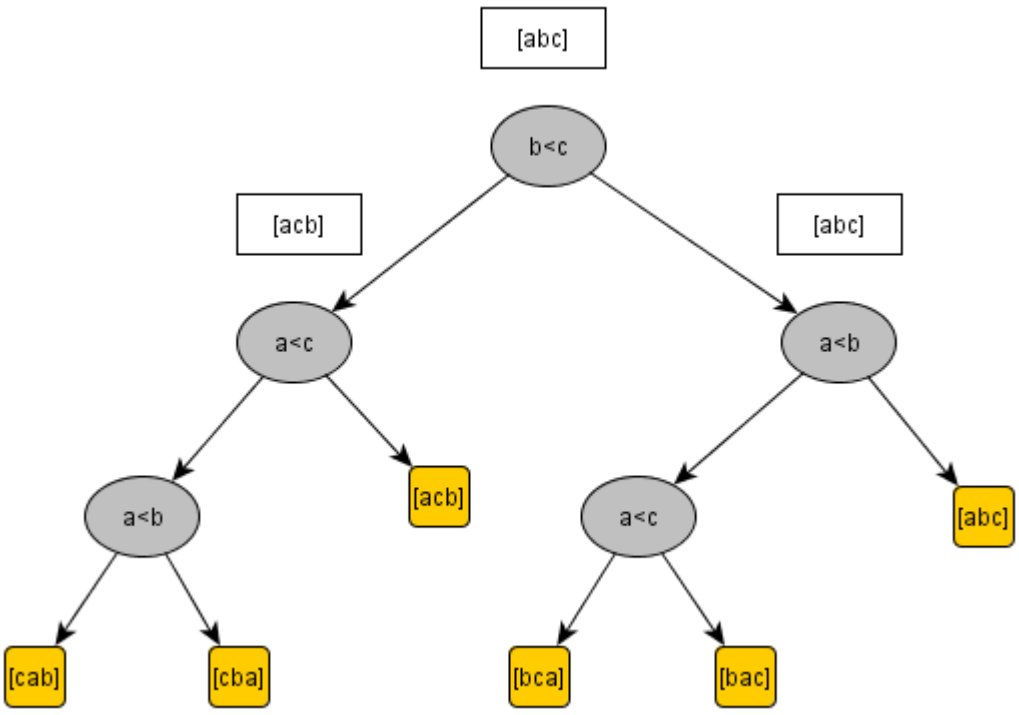
- Se utiliza como herramienta para modelar la ejecución de algoritmos que resuelvan un problema dado.

- Los nodos internos representan operaciones.
- Los externos representan el final de la ejecución de un algoritmo, cada uno para una determinada entrada.

- Por convención: Si la comparación resulta verdadera, se toma a la derecha y de lo contrario a la izquierda.

- Costo = Cantidad de operaciones para dicha entrada = Largo de dicho camino (cantidad de aristas)

ii.



iii.

Debido a que el cálculo de la parte **a-ii** se hizo para *n* potencia de dos y 3 no es potencia de dos no aplica indicar qué rama cumple.

A continuación se muestra en rojo y resaltado el camino para la elección de un peor caso.