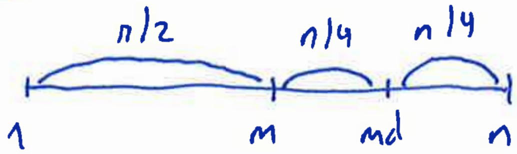
1



Además, dado que hay un bucle for cuyo orden de eficiencia es $O(n)$, la ecuación de recurrencias es

$$T(n) = T(n/2) + 2T(n/4) + n$$

Vamos a resolverla con el cambio de variable

$n = 2^h$; $h = \log_2 n$

$$T(2^h) = T(2^{h-1}) + 2T(2^{h-2}) + 2^h$$

$$T(2^h) = t_h$$

$$t_h = t_{h-1} + 2t_{h-2} + 2^h$$

$$\underbrace{t_h - t_{h-1} - 2t_{h-2}}_{\downarrow} = \underbrace{2^h}_{} \rightarrow \begin{array}{l} b = 2 \\ g(h) = 1 \\ \text{grado} = 0 \end{array} \rightarrow (x-2)$$

$$p(x) = x^2 - x - 2 \rightarrow \frac{1 \pm \sqrt{1+8}}{2} = \frac{1 \pm 3}{2} \begin{array}{l} \nearrow 2 \\ \searrow -1 \end{array} \rightarrow (x-2)(x+1)$$

Por tanto, $(x-2)^2(x+1)$

$$t_h = C_{10} 2^h + C_{11} 2^h \cdot h + C_{20} (-1)^h$$

$$T(n) = C_{10} 2^{\log_2 n} + C_{11} 2^{\log_2 n} \cdot \log_2 n + C_{20} (-1)^{\log_2 n}$$

$$T(n) = C_{10} \cdot n + C_{11} \cdot n \cdot \log_2 n + C_{20} (-1)^{\log_2 n}$$

$$T(n) \in O(n \cdot \log_2 n)$$

(3) El mayor número de parcelas consecutivas donde se puede construir puede estar en la mitad izquierda del array P (primer caso), en la mitad derecha (segundo caso), o solapándose con ambas mitades (tercer caso).

Los dos primeros casos se resuelven recursivamente. Para el tercer caso, se determina si el último elemento de la mitad izquierda y el primer elemento de la mitad derecha son iguales a 1. En tal caso, se recorren las dos mitades desde el centro mientras los elementos consecutivos sean iguales a 1.

El resultado es el máximo entre la subsecuencia máxima de la mitad izquierda, la subsecuencia máxima de la mitad derecha, y la solapada.

El caso base se da cuando el array P tiene un único elemento.

El pseudocódigo del algoritmo es:

```
int parcelas ( P, inicio, fin) {
        if (inicio == fin)      //caso base
            return P[inicio];
        else{
          mitad = (inicio+ fin)/2;
          parcelas_izq = parcelas (P, inicio, mitad);
          parcelas_der = parcelas (P, mitad+1, fin);
          if (P[mitad] == 1 && P[mitad+1] == 1){    // tercer caso
              parar = False;
              i = mitad - 1;
              suma_izq = 1;
              while (!parar && i >= inicio){
                  if ( P[i] == 1){
                      i--;
                      suma_izq++;
                  }
                  else
                      parar = true;
              }
              parar = false;
              i = mitad + 2;
              suma_der = 1;
              while (!parar && i <= fin){
                  if (P[i] == 1){
                      i++;
                      suma_der++;
                  }
                  else
                      parar = true;
              }
              parcelas_med = suma_izq + suma_der;
          }
          else
              parcelas_med = 0;
          return max (parcelas_izq, parcelas_der, parcelas_med);
        }
}
```

La llamada inicial sería parcelas (P, 0, n-1), siendo n el tamaño del array P.

(2) Vamos a usar un algoritmo de vuelta atrás. Para ello, vamos a representar la solución como una tupla

$$X = (x_1, x_2, \dots, x_n)$$

donde cada x_i tendrá un valor de 0, para indicar que no se selecciona el contenedor i, o de 1, para indicar que sí se selecciona el contenedor i.

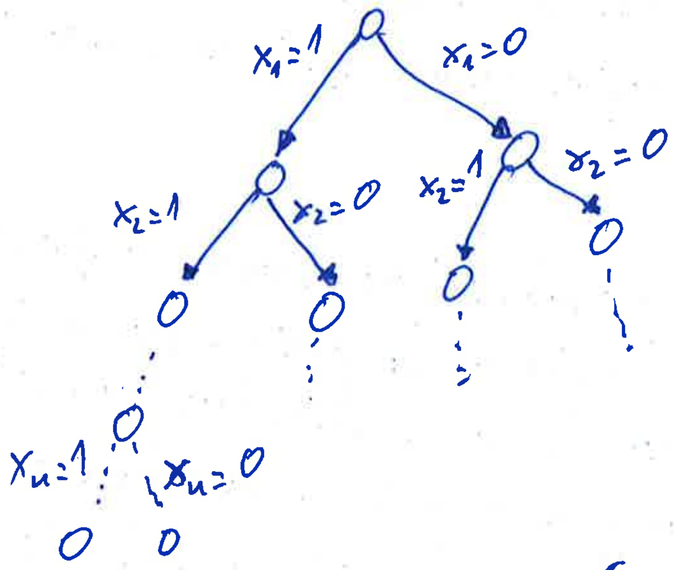
Las restricciones explícitas son:

$$x_i \in \{0, 1\} \qquad 1 \leq i \leq n$$

Las restricciones implícitas son:

$$\sum_{i=1}^{n} x_i \cdot c_i \leq N$$

El árbol de estados asociado a esta representación es



Árbol binario de profundidad n. En cada nivel i, se decide si se escoge (1) o no (0) el contenedor i.

El pseudocódigo del algoritmo es:

```
void contenedores(x, c, mejor_solucion, n, N, peso_actual, mejor_peso, k){
    if (K==n){ // estamos en un nodo hoja
        if (peso_actual > mejor_peso){
            mejor_peso = peso_actual;
            for (i=0; i<n; i++)
                mejor_solucion[i] = x[i];
        }
    }
    else{
        x[K]=1; // selecciono el contenedor K
        if (factible(c, n, N, peso_actual, K){
            peso_actual += c[K];
            contenedores(x, c, mejor_solucion, n, N, peso_actual, mejor_peso, K+1);
            peso_actual -= c[K]; // para vuelta atrás
        }
        x[K]=0; // no elijo el contenedor K
        contenedores(x, c, mejor_solucion, n, N, peso_actual, mejor_peso, K+1);
    }
}
```

mejor_solucion guarda la mejor solución encontrada, mejor_peso es el valor de la mejor solución encontrada. La variable peso_actual se usa para ir acumulando el peso de los contenedores escogidos. La variable K indica por qué componente de la solución voy (nivel de profundidad del árbol). La función factible es:

```
bool factible(c, n, N, peso_actual, K){
    if (peso_actual + c[K] <= N)
        return true;
    else
        return false;
}
```

La llamada inicial sería

peso_actual = 0;

mejor_peso = 0;

K = 0;

contenedores(x, c, mejor_solucion, n, N, peso_actual, mejor_peso, k);

Nota: en la representación de la solución se han considerado los índices de los componentes de 1 a n. En el pseudocódigo, los índices del vector que representa la solución consideramos de 0 a n-1, por eso K empieza en 0.

4. Es igual al problema de la mochila 0/1 pero teniendo en cuenta que tenemos múltiples copias de los objetos.

Las decisiones a tomar son cuántos sacos del fertilizante 1 incluyo en el camión, cuántos sacos del fertilizante 2, etc.

Para demostrar el principio de optimalidad, sea $x_1, \dots, x_n$ la solución óptima del problema con n tipos de fertilizantes y capacidad máxima del camión M kilogramos.

Hay que demostrar que $x_1, \dots, x_{n-1}$ es la solución óptima para un problema con $n-1$ tipos de fertilizantes y capacidad máxima de carga $M - x_n \cdot p_n$. Si no fuese cierto, entonces existiría una solución $y_1, \dots, y_{n-1}$ de modo que $\sum_{i=1}^{n-1} y_i \cdot p_i \leq M - x_n \cdot p_n$ y $\sum_{i=1}^{n-1} x_i b_i < \sum_{i=1}^{n-1} y_i b_i$. Pero entonces, definiendo $y_n = x_n$ obtendríamos que $\sum_{i=1}^{n-1} y_i p_i + y_n \cdot p_n \leq M$. Por tanto, tenemos una solución para el problema original y, además, esa solución cumple que $\sum_{i=1}^{n} x_i b_i < \sum_{i=1}^{n} y_i b_i$, y sería mejor que la solución óptima.

Sea $m(i,j)$ el máximo beneficio para un camión de capacidad j donde solo se usan los i primeros tipos de fertilizantes. El problema original es $m(n, M)$.

La recurrencia es

$$m(i,j) = \max_{0 \leq x_i \leq j/p_i} \left(x_i b_i + m(i-1, j - x_i p_i) \right)$$

Si selecciono x_i ~~unid.~~ sacos del fertilizante i obtengo un beneficio de $x_i b_i$ más el beneficio conseguido para un problema con $i-1$ tipos de fertilizantes y un camión de capacidad $j - x_i p_i$. El valor x_i puede variar entre 0 (no selecciono ningún saco del fertilizante de tipo i) y j/p_i (parte entera) pues no caben más de esa cantidad de sacos.

Los casos base de la recurrencia son:

$m(i,0) = 0 \quad \forall i$. Si no cabe nada, el beneficio es nulo.

$m(0,j) = 0 \quad \forall j$. Si no hay fertilizantes, no hay beneficio

El algoritmo que construye la tabla de tamaño $n+1 \times M+1$ es el siguiente, donde $v(i,j)$ se usa para almacenar el número de sacos del fertilizante de tipo i seleccionados.

```
for i=0 to n
    m(i,0)=0;
for j=1 to M
    m(0,j)=0;
for i=1 to n
    for j=1 to M
        m(i,j)=m(i-1,j);
        v(i,j)=0;
        lim = j/p(i);
    for K=1 to lim                          - (simbolo del menos)
        if ( K*b(i)+m(i-1, j-K*p(i)) > m(i-1,j))
            m(i,j)=K*b(i)+m(i-1,j-K*p(i));
            v(i,j) = K;
return m(n,M)
```

Para el caso propuesto, las tablas m y v que crea el algoritmo son:

Tabla m:

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	7	7	7	14	14	14	21	21
2	0	0	0	7	8	8	14	15	16	21	22
3	0	0	0	7	8	8	14	15	16	21	22

Tabla v

	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	1	1	1	2	2	2	3	3
2	0	0	0	0	1	1	0	1	2	0	1
3	0	0	0	0	0	0	0	0	0	0	0

El número de sacos num(i) de cada tipo de fertilizante seleccionados se obtiene a partir de la tabla v del siguiente modo

```
i=n;
j=M;
while (i>1) {
   num(i)=v(i,j);
   j=j-v(i,j)*p(i);
   i=i-1;
}
```

(5) Con el algoritmo de Dijkstra. Explicar como en las diapositivas del tema 3.