UNIVERSIDAD DE GRANADA

Algorítmica

Curso 2023

Práctica 4:

Algoritmos Greedy: Problema de repartir el dinero

Amador Carmona Méndez

Miguel Ángel López Sánchez

Índice:

1. Diseño de componentes de Programación Dinámica.

1. Definición del estado: El estado en este problema se puede definir utilizando dos variables: el índice de la empresa actual y la cantidad de dinero disponible. Definimos el estado como DP(i, x), donde i representa el índice de la empresa y x representa la cantidad de dinero disponible.

2. Función objetivo: La función objetivo es maximizar el beneficio total obtenido. Podemos definir una función objetivo recursiva, DP(i, x), que representa el máximo beneficio que se puede obtener considerando las empresas desde 1 hasta i y con un presupuesto de x.

3. Ecuación recurrente: La ecuación recurrente nos permite calcular el valor del estado actual basado en los valores de los estados anteriores. La ecuación recurrente para este problema se puede definir de la siguiente manera:
DP(i, x) = max(DP(i-1, x), DP(i-1, x-pi-ci) + bi*pi), donde i es el índice actual, x es la cantidad de dinero disponible, pi es el precio de la acción, ci es la comisión por acción y bi es el beneficio esperado por acción.

4. Inicialización: Inicializamos los valores de DP(i, x) para el caso base, donde i = 0 o x = 0.

5. Recorrido de los estados: Recorremos los estados en un orden adecuado (generalmente de manera incremental) para calcular los valores de DP(i, x) utilizando la ecuación recurrente.

6. Recuperación de la solución: Después de calcular los valores de DP(i, x), podemos recuperar la solución óptima rastreando los estados anteriores y seleccionando las acciones correspondientes.

2. Diseño e implementación del algoritmo básico (fuerza bruta) a partir de la ecuación recurrente de forma directa.

Siendo:

X: Cantidad de dinero disponible para invertir.
N: Número total de empresas disponibles.
a: Lista de tamaño N con el número total de acciones disponibles para cada empresa.

p: Lista de tamaño N con el precio de cada acción.
b: Lista de tamaño N con el beneficio esperado por cada acción (en porcentaje).
c: Lista de tamaño N con la comisión por acción para cada empresa.

C/C++

```
 Función FuerzaBruta(X, N, a, p, b, c):
   mejorBeneficio = 0
   mejorComb = []

   Para cada combinación en todasLasCombinaciones(N):
       dineroRestante = X
       beneficioTotal = 0
       combinacionActual = []

       Para i desde 0 hasta N-1:
           accionesCompradas = min(a[i], dineroRestante //
(p[i] + c[i]))
           dineroRestante -= accionesCompradas * (p[i] +
c[i])
           beneficioTotal += accionesCompradas * b[i] * p[i]
           combinacionActual.agregar(accionesCompradas)

       Si beneficioTotal > mejorBeneficio:
           mejorBeneficio = beneficioTotal
           mejorComb = combinacionActual

   Devolver mejorComb
```

3. Diseño de algoritmo de Programación Dinámica de acuerdo a las componentes diseñadas en el apartado 1.

Siendo:

X: Cantidad de dinero disponible para invertir.
N: Número total de empresas disponibles.

a: Lista de tamaño N con el número total de acciones disponibles para cada empresa.
p: Lista de tamaño N con el precio de cada acción.
b: Lista de tamaño N con el beneficio esperado por cada acción (en porcentaje).
c: Lista de tamaño N con la comisión por acción para cada empresa.

```
Unset
Función ProgramaciónDinámica(X, N, a, p, b, c):
    // Inicialización
    DP = Matriz de tamaño (N+1) x (X+1) inicializada con 0

    Para i desde 1 hasta N:
        Para x desde 1 hasta X:
            // Casos base
            Si i = 0 o x = 0:
                DP[i][x] = 0
            Si a[i-1] <= x:
                // Ecuación recurrente
                DP[i][x] = max(DP[i-1][x], DP[i-1][x - (p[i-1]
+ c[i-1])] + b[i-1] * p[i-1])
            Sino:
                DP[i][x] = DP[i-1][x]

    // Recuperación de la solución óptima
    mejorBeneficio = DP[N][X]
    combinacionOptima = []
    x = X

    Para i desde N hasta 1:
        Si DP[i][x] != DP[i-1][x]:
            accionesCompradas = min(a[i-1], x // (p[i-1] +
c[i-1]))
            combinacionOptima.agregar(accionesCompradas)
            x -= accionesCompradas * (p[i-1] + c[i-1])

    Devolver combinacionOptima
```

4. Implementación del algoritmo de Programación Dinámica.

```
C/C++
vector<int> ProgramacionDinamica(int X, int N, vector<int>& a, vector<int>& p,
vector<int>& b, vector<int>& c) {
    // Inicialización
    vector<vector<int>> DP(N + 1, vector<int>(X + 1, 0));

    for (int i = 1; i <= N; i++) {
        for (int x = 1; x <= X; x++) {
            // Casos base
            if (i == 0 || x == 0) {
                DP[i][x] = 0;
            } else if (a[i - 1] <= x) {
                // Ecuación recurrente
                DP[i][x] = max(DP[i - 1][x], DP[i - 1][x - (p[i - 1] + c[i - 1])] +
b[i - 1] * p[i - 1]);
            } else {
                DP[i][x] = DP[i - 1][x];
            }
        }
    }

    // Recuperación de la solución óptima
    int mejorBeneficio = DP[N][X];
    vector<int> combinacionOptima;
    int x = X;

    for (int i = N; i >= 1; i--) {
        if (DP[i][x] != DP[i - 1][x]) {
            int accionesCompradas = min(a[i - 1], x / (p[i - 1] + c[i - 1]));
            combinacionOptima.push_back(accionesCompradas);
            x -= accionesCompradas * (p[i - 1] + c[i - 1]);
        }
    }

    return combinacionOptima;
}
```

5. Cálculo de eficiencia del algoritmo básico y de Programación Dinámica.

```
Función ProgramaciónDinámica(X, N, a, p, b, c):
    // Inicialización
    DP = Matriz de tamaño (N+1) x (X+1) inicializada con 0

    Para i desde 1 hasta N:
        Para x desde 1 hasta X:
            // Casos base
            Si i = 0 o x = 0:
                DP[i][x] = 0
            Si a[i-1] <= x:
                // Ecuación recurrente
                DP[i][x] = max(DP[i-1][x], DP[i-1][x - (p[i-1]
+ c[i-1])] + b[i-1] * p[i-1])
            Sino:
                DP[i][x] = DP[i-1][x]

    // Recuperación de la solución óptima
    mejorBeneficio = DP[N][X]
    combinacionOptima = []
    x = X

    Para i desde N hasta 1:
        Si DP[i][x] != DP[i-1][x]:
            accionesCompradas = min(a[i-1], x // (p[i-1] +
c[i-1]))
            combinacionOptima.agregar(accionesCompradas)
            x -= accionesCompradas * (p[i-1] + c[i-1])

    Devolver combinacionOptima
```

$O(1)$

$+$

$O(X)$ $O(N) \Rightarrow O(X*N)$

Algoritmo PD

$+$

$O(N)$

Eficiencia del algoritmo:

Para analizar la eficiencia del algoritmo, la primera parte tiene la mayor complejidad del algoritmo ya que en el peor de los casos sería $O(N*X)$, ya que el bucle anidado recorre la matriz de la programación Dinámica con 'N' filas y 'X' columnas.

La segunda parte del algoritmo, es decir, el segundo bucle, se encarga de recuperar la solución <u>OPTIMA</u>, del algoritmo.

Por lo tanto, la eficiencia del algoritmo sería:

$O(N*X + 1 + N + 1)$; Aplicando la regla de la suma quedaría como: $O(N*X)$ ya que estamos fijándonos en la eficiencia en el peor de los casos.

```
Función FuerzaBruta(X, N, a, p, b, c):
  mejorBeneficio = 0
  mejorComb = []

  Para cada combinación en todasLasCombinaciones(N):
      dineroRestante = X
      beneficioTotal = 0
      combinacionActual = []

      Para i desde 0 hasta N-1:
          accionesCompradas = min(a[i], dineroRestante //
(p[i] + c[i]))
          dineroRestante -= accionesCompradas * (p[i] +
c[i])
          beneficioTotal += accionesCompradas * b[i] * p[i]
          combinacionActual.agregar(accionesCompradas)

      Si beneficioTotal > mejorBeneficio:
          mejorBeneficio = beneficioTotal
          mejorComb = combinacionActual

  Devolver mejorComb
```

$O(2^{N})$

$O(N)$

Algoritmo Fuerza Bruta

en este caso, hacemos una combinación de todas las posibles combinaciones por lo que es 2^n posibilidades.

Eficiencia del Algoritmo

Este algoritmo a simple vista no es muy eficaz ya que desde un principio lo que hace es una combinación de todas y cada una de las compras posibles de las acciones (en este caso 'N'). El porque es 2^n y no otro, es porque en este caso tenemos 'N' acciones y en cada acción dos posibilidades, comprar ó no comprar.

Esto hace bastante parecido al ejercicio de la mochila booleana (1 ó 0) en el que podíamos introducir un objeto ó no, en ella, maximizando su valor y con el mínimo peso

En resumen, este algoritmo tiene una eficiencia en el peor de los casos de $O(2^n * N)$ ya que para cada combinación generada por cada acción, se combina con el resto de acciones, contando las dos posibilidades, tanto como comprarla, como no comprarla.

Conclusion: Es mas eficiente el algoritmo PD que el F.Bruta

6. Aplicación a dos instancias de problema concretas.

Código del caso1:

```cpp
int main() {
    int X = 1000;  // Cantidad de dinero disponible
    int N = 10;    // Número total de empresas

    // Ejemplo de datos de empresas
    vector<int> a = {10, 20, 15, 12, 8, 18, 7, 14, 9, 11};  //
Número total de acciones
    vector<int> p = {50, 30, 25, 40, 20, 35, 45, 60, 55, 50};
// Precio de las acciones
    vector<int> b = {10, 15, 12, 8, 5, 9, 7, 14, 11, 13};  //
Beneficio esperado en porcentaje
    vector<int> c = {5, 3, 4, 2, 2, 4, 3, 5, 4, 3};  //
Comisión por acción

    vector<int> combinacionOptima = ProgramacionDinamica(X, N,
a, p, b, c);

    // Imprimir la combinación óptima de acciones compradas
    cout << "Combinación óptima de acciones compradas: ";
    for (int i = 0; i < combinacionOptima.size(); i++) {
        cout << combinacionOptima[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Salida del caso1:

```
Combinación óptima de acciones compradas:
Empresa 1: 10 acciones
Empresa 2: 20 acciones
Empresa 3: 15 acciones
Empresa 4: 12 acciones
Empresa 5: 8 acciones
Empresa 6: 18 acciones
Empresa 7: 0 acciones
```

```
Empresa 8: 0 acciones
Empresa 9: 0 acciones
Empresa 10: 0 acciones
```

Código del caso2:

```C/C++
int main() {
    int X = 1000;  // Cantidad de dinero disponible
    int N = 20;    // Número total de empresas

    // Ejemplo de datos de empresas
    vector<int> a = {10, 20, 15, 12, 8, 18, 7, 14, 9, 11, 10,
20, 15, 12, 8, 18, 7, 14, 9, 11};  // Número total de acciones
    vector<int> p = {50, 30, 25, 40, 20, 35, 45, 60, 55, 50,
50, 30, 25, 40, 20, 35, 45, 60, 55, 50};  // Precio de las
acciones
    vector<int> b = {10, 15, 12, 8, 5, 9, 7, 14, 11, 13, 10,
15, 12, 8, 5, 9, 7, 14, 11, 13};  // Beneficio esperado en
porcentaje
    vector<int> c = {5, 3, 4, 2, 2, 4, 3, 5, 4, 3, 5, 3, 4, 2,
2, 4, 3, 5, 4, 3};  // Comisión por acción

    vector<int> combinacionOptima = ProgramacionDinamica(X, N,
a, p, b, c);

    // Imprimir la combinación óptima de acciones compradas
    cout << "Combinación óptima de acciones compradas:" <<
endl;
    for (int i = 0; i < N; i++) {
        cout << "Empresa " << (i + 1) << ": " <<
combinacionOptima[i] << " acciones" << endl;
    }

    return 0;
}
```

Salida del caso2:

```Unset
Combinación óptima de acciones compradas:
Empresa 1: 10 acciones
Empresa 2: 20 acciones
Empresa 3: 15 acciones
Empresa 4: 12 acciones
Empresa 5: 8 acciones
Empresa 6: 18 acciones
Empresa 7: 7 acciones
Empresa 8: 14 acciones
Empresa 9: 9 acciones
Empresa 10: 11 acciones
Empresa 11: 10 acciones
Empresa 12: 20 acciones
Empresa 13: 15 acciones
Empresa 14: 12 acciones
Empresa 15: 8 acciones
Empresa 16: 18 acciones
Empresa 17: 7 acciones
Empresa 18: 14 acciones
Empresa 19: 9 acciones
Empresa 20: 11 acciones
```