Universidad de Concepción

Facultad de ingeniería
Departamento de ingeniería informática y Ciencias de la computación

Tarea 3 de Análisis de Algoritmos

Programación dinámica

Lisette Morales.

Pablo Venegas.

Profesor : Andrea Rodríguez.

Índice

1. Problema asignado:

Usted en conjunto con su mejor amigo han desenterrado una bolsa que contiene M monedas de diverso valor (1 ¡M ¡1000). Lamentablemente ambos no poseen buenas habilidades matemáticas por lo se les dificulta el lograr una división que los satisfaga a ambos. Por suerte sus conocimientos en programación si son de ar por lo que han decidido implementar un algoritmo capaz de lograr una división que minimice la diferencia en caso de no existir una repartición equitativa. Se le pide demostrar que lo implementado es correcto y que complejidad posee. Asuma que el valor de cada moneda en la bolsa vara entre 1 y 1000.

2. Descripción de la solución

Primero se crea un mapa que será el que almacene la solución a los subproblemas, este es designado global dado que por el tipo de implementación recursiva ocupado es lo mejor para este problema e implementación.

La función recibe como parámetros todos los elementos dentro del vector, que en este caso es la bolsa con las monedas, la cantidad de monedas y las sumas acumuladas de ambas personas.

El caso base para la función es que cuando la bolsa está vacía, retorna el valor absoluto de la diferencia entre las dos sumas acumuladas.

Luego construye una única "key" para el mapa para diferenciar de manera única cada subproblema y así no repetir un cálculo ya realizado.

Luego si el subproblema es visto por primera vez, este es calculado siguiendo dos casos. En el primer caso se incluye en la suma acumulada de la primera persona el valor de la moneda n y se ejecuta de manera recursiva, en el segundo caso se excluye de la primera persona y se le otorga a la otra, de esta forma así se analizan todas las posibles configuraciones distintas para encontrar menor diferencia entre los valores sin tener que volver a resolver subproblemas ya resueltos, luego la función retorna la menor diferencia entre estas dos "implementaciones" de manera recursiva del algoritmo, obteniendo así la respuesta a nuestro problema, cabe destacar que la función retorna 0 si es que se encuentra que una de las soluciones conlleva a que ambas sumas acumuladas sean iguales, por ende alcanzando la "máxima satisfacción" posible para estos dos amigos repartiendo las monedas encontradas.

3. Implementación

```
unordered_map<string, int> mapa;

int minParticion(vector <int> Monedas, int cantidadMonedas, int Suma1, int Suma2)
{
  if (cantidadMonedas < 0){return abs(Suma1 - Suma2);}

  string distanciaMin = to_string(cantidadMonedas) + "|" + to_string(Suma1);

  if (mapa.find(distanciaMin) == mapa.end())
  {
    int inc = minParticion(Monedas, cantidadMonedas - 1, Suma1 + Monedas[cantidadMoned

    int exc = minParticion(Monedas, cantidadMonedas - 1, Suma1, Suma2 + Monedas[cantid

    mapa[distanciaMin] = min (inc, exc);
  }
  return mapa[distanciaMin];
}
```

3.1. Main

Principalmente en nuestro main, lo único que se hace es crear nuestra bolsa de monedas dado un parámetro M que representa la cantidad total de monedas encontradas en la bolsa, ingresado como argumento; si se encuentra dentro de los límites establecidos, y se le asignan valores aleatorios a todas las monedas contenidas en la bolsa, luego se procede a llamar a la función que calcula la diferencia mínima que se puede obtener al repartir las monedas.

```
int main(int argc, char const *argv[]){
  int M = atoi(argv[1]);
  if (M <= 1 || M >= 1000){
    cout << "Nada que hacer aqui, compilacion terminada por valor de M fuera de los li
    return 0;
  }

vector <int> monedas;

random_device rd;
  mt19937 eng(rd());
  uniform_int_distribution<> distr(1, 1000);

  for(int i=0; i<M; ++i) monedas.push_back(distr(eng));

//  sort(monedas.begin(),monedas.end());
/*
for (int i = 0; i < M ; i++){
cout << monedas[i] << " ";
}
cout << endl;
*/
int minDiferencia = 0;
//  TIMERSTART(minimo)
minDiferencia = minParticion(monedas, M-1, 0, 0);
//  TIMERSTOP(minimo)
  cout << "La diferencia minima es: " << minDiferencia<<endl;

return 0;
}
```

4. Análisis de correctitud

Para el problema entregado el caso base de este es cuando se nos entregan 2 monedas, y se debe decidir a quien entregarlas, en este caso, a ambas personas se le entrega una de las monedas, de esta forma cada moneda es entregada a una persona, por ende este caso base se cumple y es correcto dado que se le entrega la cantidad total de monedas de manera equitativa, de esta forma esto pasa a ser una solo una subestructura del problema,cuando se analiza el caso para n monedas a repartir, se debe calcular como repartir estas n monedas de manera que se obtenga la mínima diferencia entre las sumas acumuladas de las monedas que cada persona posee, por ende esta decisión se basa principalmente en la decisión que se realizó en el paso n-1 y con esta solución decidir a quien cederle la moneda "n" para que se cumpla que se debe encontrar la mínima diferencia entre estas dos sumas acumuladas, por ende dado que siempre se utiliza la decisión anterior para decidir la nueva decisión a tomar, se puede decir que nuestro problema se puede resolver mediante Programación Dinámica, y dado que nuestro caso base es una subestructura óptima de solución del problema, y con la siguiente recurrencia que se tiene.

F(N,S1,S2) = minF(N-1,S1+Vn,S2), F(N-1,S1,S2+Vn)

Donde: F = función de recurrencia que describe nuestro problema

N = número de monedas a repartir
S1 = suma acumulada del individuo 1
S2 = suma acumulada del individuo 2
Vn = Valor de la moneda a repartir

Por ende dado que nuestro caso base corresponde a una subestructura óptima de nuestro problema y dado que para tomar la decisión en el caso n se requiere de la decisión tomada en el caso anterior, comprobamos que este problema es DP y se soluciona dado la subestructura anterior planteada.

5. Análisis asintótico

Para este algoritmo la complejidad que alcanza es de O(nxsum) dado que nuestro algoritmo es recursivo y al generar esta recursividad se tiene que operar sobre todos los conjuntos posibles de solución, dado que el problema se resuelve por programación dinámica hay subproblemas que pueden omitirse dado que estos fueron previamente calculados pero en el peor caso bajo estas condiciones se alcanza esa complejidad.