Practica 3: Voraces y Programación Dinámica

En esta practica vamos a tratar dos tipos de problemas con algoritmos voraces y de programación dinámica, el primer ejercicio o problema a realizar es conseguir llegar a través de la suma de los valores de un conjunto de valores que nos proporcionan, este problema se realizará con PD y con Voraces. El segundo problema a realizar es conseguir un valor a través de un conjunto de elementos con las operaciones básicas de suma, resta, multiplicación y división, este se realizará solo con PD.

EJERCICIO SUMA HASTA UN NÚMERO VORAZ:
El primer ejercicio como he dicho con anterioridad consiste en que dado un conjunto de números dar el resultado igual o más cercano al valor que buscamos. Los candidatos a utilizar es el conjunto de valores que podemos usar para sumar y conseguir llegar al valor final que buscamos, en nuestro caso serán los archivo de datos que se nos han proporcionado en la practica. La función de solución son aquellos valores que vamos añadiendo a un vector y que son aquellos valores que nos hemos quedado para realizar la suma y esta función general la especifico luego con una captura. La función de selección nos va a seleccionar entre los valores posibles, aquel valor máximo del conjunto que no se pase del objetivo al sumarlo al valor que llevamos hasta el momento, también adjuntare el código luego. La función factible es la comprobación de si podemos añadir el elemento en nuestro conjunto de solución o no, básicamente lo que comprueba es que si sumamos ese valor a lo que llevamos acumulado hasta ahora no se pasa del objetivo. La función o estado objetivo no es más que el valor al que queremos llegar, luego lo que buscamos es intentar acercarnos al máximo a este valor. Adjunto capturas del código que representa todo esto:

```
int Seleccion(vector<int>& c){
  int n = c.size();
  int max = numeric_limits<int>::min();
  int pos = -1;

  for (int i=0;i<n;i++){
    if(max<c[i]){
      max = c[i];
      pos = i;
    }
  }
  c.erase(c.begin()+pos);
  return max;
}

bool Factible(int x, int s, int M){
  if(s+x<=M)
    return true;

  return false;
}

pair<vector<int>,int> MaximoVoraz(vector<int> S, int M){
  pair<vector<int>,int> resultado;
  vector<int> c = S;
  int cantidadActual=0;
  int x=0;
  vector<int> s;
  while(cantidadActual < M && c.size()>0){
    x = Seleccion(c);
    if(Factible(x,cantidadActual, M)){
      s.push_back(x);
      cantidadActual+=x;
    }
  }

  resultado.first = s;
  resultado.second = cantidadActual;

  return resultado;
}
```

Como se puede observar la función Selección simplemente va a buscar el valor máximo de nuestro conjunto de valores posible a sumar y la función Factible no es más que la comprobación si la suma de este valor es posible de usarla o no, es decir, si se pasa del valor objetivo o no. En cuanto a la función solución, lo que se realiza es un bucle mientras que la cantidad acumulada actual sea menor que la objetiva y mientras que haya valores posible para coger lo que se realiza es buscar el valor mayor de nuestro conjunto de valores, comprobamos si es factible o no y si lo es lo añadimos a nuestro conjunto solución final. En cuanto a la eficiencia teórica de este algoritmo, en cuanto a la selección es de O(n) y luego el bucle while es lo único que puede verse en la eficiencia, ya que la función factible es una comprobación de O(1), luego el orden teórico por el que se maneja este algoritmo es de O(n*logn).

EJERCICIO SUMA HASTA UN NÚMERO PROGRAMACIÓN DINÁMICA:

En este caso es el mismo problema que el anterior solo que se aborda con programación dinámica, para ello debemos rellenar una tabla con esta técnica y finalmente hay que reconstruir la solución a través del uso de la tabla. La ecuación recurrente obtenida de este problema es la siguiente:

$$\text{Suma(k,m)} = \begin{cases} 0, m = 0 \\ 0, k = 0 \\ -INF, k < 0 \cup m < 0 \\ max(Suma(k-1, m), Suma(k-1, m-k)+k) \end{cases}$$

Este algoritmo genera una tabla con tantas filas como valores tengamos en el conjunto que utilicemos y tantas columnas como el valor objetivo al que queremos llegar. Se rellena filas por columnas teniendo en cuenta la ecuación recurrente anterior, es decir, se van visitando cada una de las posiciones de la tabla y se aplica la ecuación recurrente anterior. Lo más importante de la ecuación recurrente anterior es el maximo entre echar el elemento en el conjunto final o no, si no lo echas te quedarías con el valor que ya tenias y ya tienes que comprobar un valor menos y si lo echas pues eliminas el valor que vamos a utilizar y aumentamos el valor acumulado con el valor que hemos elegido, las otras sentencias de la ecuación recurrente es para tratar limites y que no de fallos en la implementación. En cuanto a la reconstrucción final de la solución consiste en irse al ultimo valor de la ultima fila y columna de la tabla y a partir de ahi si el valor que tiene justo arriba, en la fila superior, tienen un valor igual no cogemos ese valor y subimos a la fila superior, si estos dos valores son diferentes cogemos ese valor, subimos a la fila de arriba y nos vamos en esa fila a la izquierda tantas veces como nos diga el valor que hemos elegido, y así hasta que lleguemos a un valor de 0, adjunto código que he implementado a continuación.

```
vector<int> rellenarTabla(vector<vector<int> > &tabla, vector<int> c, int obj) {
    for(unsigned int i = 0; i< tabla[0].size(); i++) {
        tabla[0][i] = 0;
    }

    for(unsigned int i = 0; i < tabla.size(); i++) {
        for( unsigned int j = 0; j < tabla[i].size(); j++) {
          if(i==0 || c[i]==0){
            tabla[i][j] = 0;
          }else if(i<0 || c[i]<0){
            tabla[i][j] = numeric_limits<int>::min();
          }else if (c[i] > int(j)) {
              tabla[i][j] = tabla[i-1][j];
          }else{
            tabla[i][j] = max(tabla[i-1][j], c[i] + tabla[i-1][j-c[i]]);
          }
        }
    }

    vector <int> solucion;

    int i,j;
    i = c.size()-1;
    j = obj;

    //recomponer la solución
    while(j != 0 && i != 0) {
        if(tabla[i-1][j] == tabla[i][j]) {
            i = i -1;
        } else {
            solucion.push_back(c[i]);
            j = j-c[i];
            i = i -1;
        }
    }

    return solucion;
}
```

Como podemos observar, el método rellenarTabla es un método que va a realizar todo lo que he dicho anteriormente, es decir, rellena la tabla y devuelve un vector con los elementos que hemos usado para acercarnos o llegar al valor objetivo. El bucle for realiza el relleno de la tabla según la ecuación recurrente y en bucle while final recompone la solución a través de la consulta a la tabla y realizando el procedimiento que he explicado anteriormente. En cuanto a la eficiencia de este algoritmo el primer bucle for llega a un orden de O(n*m) siendo n las filas, que en nuestro caso es el numero de valores que tenemos en el conjunto de posibles números a usar y el valor de m son las columnas que puede variar según el problema, luego la eficiencia será de O(n*m).

EJERCICIO OPERACIONES BÁSICAS HASTA UN NÚMERO PROGRAMACIÓN DINÁMICA:

En este ejercicio lo que se va a realizar es que a través de un conjunto de números y un valor objetivo tenemos que llegar a este a través de las operaciones básicas de suma, resta. multiplicación y división, para ello se utiliza la técnica de programación dinámica rellenando su tabla correspondiente según una ecuación recurrente que dejare a continuación. Después de rellenar la tabla para conseguir los resultados tan solo hay que reconstruir la solución a través del procedimiento que antes explique en el anterior ejercicio. La ecuación que he utilizado es la siguiente:

$$A(k,m)=\begin{cases} min(A(k-1,m),A(k-1,m-k),A(k-1,m+k),A(k-1,m/k),A(k-1,m*k)) \\ 0, k=0 \vee m=0 \\ m, k=0 \wedge m>0 \\ +INF, m<0 \end{cases}$$

La tabla que vamos a rellenar va a tener un tamaño de tantas filas como valores tengamos en el conjunto y tantas columnas como valores del conjunto tengamos por el valor al que queramos llegar. En cuanto al relleno de la tabla se va a realizar por filas y a su ve por columnas, se van visitando cada una de ellas y se realiza la ecuación recurrente que nos va a proporcionar el valor que buscamos. Lo importante de esta ecuación recurrente es sobre todo el mínimo entre no echar el elemento, echar sumándolo, echar restándolo, echar multiplicándolo o echarlo dividiéndolo, ya que las otras condiciones son sobre todos limites que nos podemos acceder a la matriz y demás. Para representar cuando sumo, resto… en la tabla lo que he ido introduciendo en la matriz son un tipo de dato pair con un valor entero (el valor que vamos aplicar la operación básica) y un char que represente la operación básica que queramos (+,-,*,/), así cuando reconstruimos la solución tenemos directamente la operación que debemos realizar. Para la reconstrucción de la solución aplicamos el mismo procedimiento que antes en el anterior ejercicio es exactamente igual, solo que en vez de aplicar siempre la suma, vamos aplicar la operación que este representada en la casilla en la que nos encontremos. En cuanto al algoritmo que he utilizado adjunto captura para ello:

```
vector<pair<char,int>> rellenarTabla(vector<vector<pair<char,int>>> &tabla, vector<int> c, int obj){
  int limite_max = std::numeric_limits<int>::max();
  pair<char,int> nada,suma,resta,multiplicacion,division,MINIMO;

  for(unsigned int i = 0; i< tabla[0].size(); i++) {
      tabla[0][i].second = i;
      tabla[0][i].first = ' ';
  }

  for(int i = 1; i < tabla.size(); i++) {
    for( int j = 0; j < tabla[i].size(); j++) {
      if(i==0 || c[i]==0){
        tabla[i][j].first=' ';
        tabla[i][j].second=0;
      }else if(i==0 && c[i]>0){
        tabla[i][j].first=' ';
        tabla[i][j].second=c[i];
      }else if(c[i]<0){
        tabla[i][j].first=' ';
        tabla[i][j].second=limite_max;
      }else{
        nada.first=' ';
        nada.second=tabla[i-1][j].second;

        if(j-c[i] < 0) {
          suma.first = '+';
          suma.second = limite_max;
        } else {
          suma.first = '+';
          suma.second = tabla[i-1][j-c[i]].second;
        }

        if(j+c[i] > tabla[0].size()){
          resta.first = '-';
          resta.second = limite_max;
        }else{
          resta.first = '-';
          resta.second = tabla[i-1][j+c[i]].second;
        }

        if(j*c[i] > tabla[0].size()) {
          division.first = '/';
          division.second = limite_max;
        } else {
          division.first = '/';
          division.second = tabla[i-1][j*c[i]].second;
        }

        if(j%c[i] != 0) {
          multiplicacion.first = '*';
          multiplicacion.second = limite_max;
        } else {
          multiplicacion.first = '*';
          multiplicacion.second = tabla[i-1][j/c[i]].second;
        }
```

```
vector <pair<char,int> > solucion;

int i,j;
i = c.size()-1;
j = obj;

while(j != 0 && i != 0) {
  if(tabla[i-1][j].second == tabla[i][j].second) {
    i = i -1;
  } else {
    solucion.push_back(make_pair(tabla[i][j].first, c[i]));
    if(tabla[i][j].first == '-')
      j = j+c[i];
    else if(tabla[i][j].first == '+')
      j = j-c[i];
    else if(tabla[i][j].first == '*')
      j = j/c[i];
    else if(tabla[i][j].first == '/')
      j = j*c[i];
    else
      j = j-1;

    i = i - 1;
  }
}

return solucion;
```

Como podemos observar la primera parte del código es recorrer la tabla y aplicar la ecuación recurrente, voy hacer incapié en la parte importante del mínimo entre no echar el elemento o usar una operación básica en el que tengo que destacar que la división solo se aplicará en el caso de que la división sea exacta con resto 0. Viendo el código podemos darnos cuenta que recorrer toda la tabla tiene una eficiencia de O(max(n)*m) siendo n el numero de elementos del conjunto y m el valor al que queremos llegar. Luego la eficiencia total teórica de este algoritmo es O(max(n)*m). Un ejemplo de ejecución sería el siguiente:

```
raul@raul-UX305LA:~/Escritorio$ ./PD_OperacionesBasicas numeros2.txt 2
Valor al que queremos llegar: 2
Valores posibles para sumar: 0 10 9 8

 ---> Matriz solución:

       m=0  m=1  m=2  m=3  m=4  m=5  m=6  m=7  m=8  m=9 m=10 m=11 m=12 m=13 m=14 m=15 m=16 m=17 m=18 m=19 m=20
k = 0    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19   20
k = 10   0    1    2    3    4    5    6    7    8    9   +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   *2
k = 9    0   -0   -1   -2   -3   -4   -5   -6   -7   +0    0    1    2    3    4    5    6    7   *2   +0   +1
k = 8    0    0   -0   -1   -2   -3   -4   -5   +0    0    0   -0   -1    3    4    5   *1   +0   +0    0    1

Objetivo: 2
Solucion: { 8 -  10 }
```