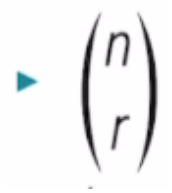
Semana 10.- Programación dinámica I

Nociones básicas de la programación dinámica

Los algoritmos de programación dinámica sirven para mejorar la eficiencia en problemas en los que hay solapamiento entre subproblemas.

Utiliza una tabla en la que se almacenan las soluciones de subproblemas ya solucionados.

Se va a utilizar para ello el calculo de números combinatorios

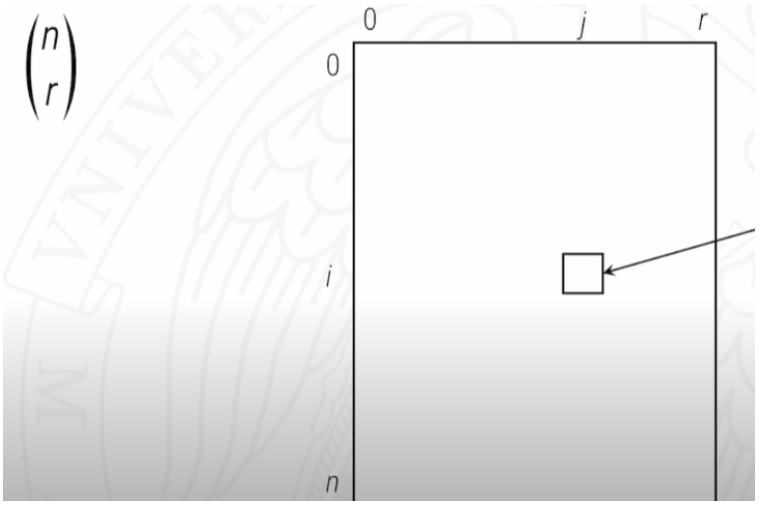


- n elementos
- cardinal r
- La operación en conjunto significa el número de subconjuntos de cardinal r que tiene un conjunto de n elementos.

Almacenamiento en programación dinámica

- Utilización de una tabla (array multidimensional) donde se almacenan subproblemas ya resueltos.
- La tabla tiene tantas dimensiones como argumentos tenga la recurrencia.
- El tamaño de cada dimensión coincide con los valores que puede tomar el argumento correspondiente.
- Cada subproblema se asocia a una posición de la tabla.

Ejemplo: tabla para un problema con 2 argumentos:



Programación dinámica descendente

- Mantiene el diseño **recursivo.**
- La función recibe como **parámetro de entrada/salida, la tabla** donde se almacenan las soluciones a subproblemas ya resueltos.
- Antes de resolver de manera recursiva un subproblema, se mira en la tabla por si ya se hubiera resuelto.
- Tras resolver un subproblema recursivo, su **solución se almacena en una tabla.**
- Es necesario **saber si un subproblema está resuelto o no.**

Ejemplo: números combinatorios

```
int num_combi(int i, int j, Matriz<int> & C) {
   if (j == 0 || j == i) return 1;
   else if (C[i][j] != -1) return C[i][j];
   else {
      C[i][j] = num_combi(i-1, j-1, C) + num_combi(i-1, j, C);
      return C[i][j];
   }
}

   Matriz<int> C(n+1, r+1, -1);

   cout << num_combi(n, r, C) << '\n';
```

parámetros:

- **int i:** Número combinatorio i.

- **int j:** Número combinatorio j.
- **Matriz<int>&C:** matriz de soluciones.

resolución:

- **Caso no recursivo**. (j==0 || j==1). Sabemos que en estos casos es 0.
- **Caso recursivo.**
  - if (C[i][j]≠-1). Entonces el caso ya ha sido resuelto.
  - else. El caso aun no ha sido resuelto por lo que:
    - Calculamos el numero combinatorio de i-1, j-1 e i-1, los sumamos y devolvemos el valor.

El resultado de la funcion está en la última posición de la matriz.

El coste algorítmico es de O(nr)

Programación dinámica ascendente

- **Cambiar el orden** en el que se resuelven los subproblemas.
- Comenzar por **resolver todos los subproblemas más pequeños** que se puedan necesitar, para ir **combinándolos** hasta llegar a resolver el problema original.
- Los **subproblemas se van resolviendo recorriéndolos de menor a mayor tamaño**.
- Todos los posibles subproblemas de **tamaño menor** tienen que ser re**sueltos antes de resolver uno de tamaño mayor.**

Ejemplo: resolver el triángulo de Pascal

	0	1	2	...	r
0	1	0	0	...	0
1	1	1	0	...	0
2	1	2	1	...	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	n	$\binom{n}{2}$	...	$\binom{n}{r}$

```
int pascal(int n, int r) {
   Matriz<int> C(n+1,r+1,0);
   C[0][0] = 1;
   for (int i = 1; i <= n; ++i) {
      C[i][0] = 1;
      for (int j = 1; j <= r; ++j)
         C[i][j] = C[i-1][j-1] + C[i-1][j];
   }
   return C[n][r];
}
```

parámetros:

- n.
- r.

resolución:

- **Caso base:** r>n solución=0.
- **Caso recursivo:**
  - Recorremos las filas de arriba a abajo y cada una de ellas de izquierda a derecha
  - Los casos bases se rellenan con 1.
  - La solución está en C[n][r].

El coste algorítmico es de O(nr)

Una de las ventajas de la programación dinámica ascendente es que proporciona **reducción de espacio** para la tabla en ocasiones.

Problema del cambio de monedas

1. **Definición del problema:**
   - Dado un conjunto de monedas M={m1,m2,...,mn} con cantidades ilimitadas, encontrar el menor número de monedas para pagar una cantidad C>0.

     M={m1,m2,...,mn}M = \{m_1, m_2, \dots, m_n\}

     C>0C > 0
2. **Características principales:**
   - No siempre funciona una estrategia voraz (usar la moneda más grande primero).
   - Las soluciones son multiconjuntos de monedas.
   - El orden en el que se consideran las monedas no afecta el resultado final.
3. **Principio de optimalidad de Bellman:**
   - Una solución óptima a un problema contiene soluciones óptimas a sus subproblemas.

Estrategias de solución

1. **Definición recursiva:**
   - Casos básicos:
     - Si C=0, no se necesitan monedas (monedas(n,0)=0).

       C=0C = 0

       monedas(n,0)=0monedas(n, 0) = 0
   - Caso recursivo:
     - monedas(n,C)=min(monedas(n−1,C),monedas(n,C−mn)+1)monedas(n, C) = \min(monedas(n-1, C), monedas(n, C-m_n) + 1)monedas(n,C)=min(monedas(n−1,C),monedas(n,C−mn)+1).
2. **Problemas y soluciones:**
   - **Subproblemas repetidos:**
     - Usar programación dinámica para evitar cálculos redundantes.
   - **Uso de tablas:**
     - Crear una matriz donde monedas[i][j] almacena el mínimo número de monedas para pagar j con las primeras i monedas.

       monedas[i][j]monedas[i][j]

Implementación

1. **Con una matriz:**

```
cpp
Copiar código
EntInf devolver_cambio(vector<int> const& M, int C) {
    int n = M.size();
    Matriz<EntInf> monedas(n+1, C+1, Infinito);
    monedas[0][0] = 0;
    for (int i = 1; i <= n; ++i) {
        monedas[i][0] = 0;
        for (int j = 1; j <= C; ++j) {
            if (M[i-1] > j)
                monedas[i][j] = monedas[i-1][j];
            else
                monedas[i][j] = min(monedas[i-1][j], mone
```

```
das[i][j - M[i-1]] + 1);
        }
    }
    return monedas[n][C];
}
```

2. **Optimización espacial:**
    - Reducir el uso de memoria a un vector unidimensional.
    - Calcular en el propio vector reutilizando datos previos.

Reconstrucción de la solución

1. **Matriz completa:**
    - Volver hacia atrás en la matriz para determinar las monedas usadas.
2. **Optimización espacial:**
    - Utilizar solo el último estado del vector para determinar las monedas seleccionadas.

Ejemplo de uso

- Monedas: M={1,2,5}, cantidad: C=8.

  M={1,2,5}M = \{1, 2, 5\}

  C=8C = 8
- Matriz calculada muestra monedas[3][8]=3 (monedas: {1,1,1}).

  monedas[3][8]=3monedas[3][8] = 3

  {1,1,1}\{1, 1, 1\}