Programación dinámica

April 24, 2018

Contenidos

Sucesión de Fibonacci

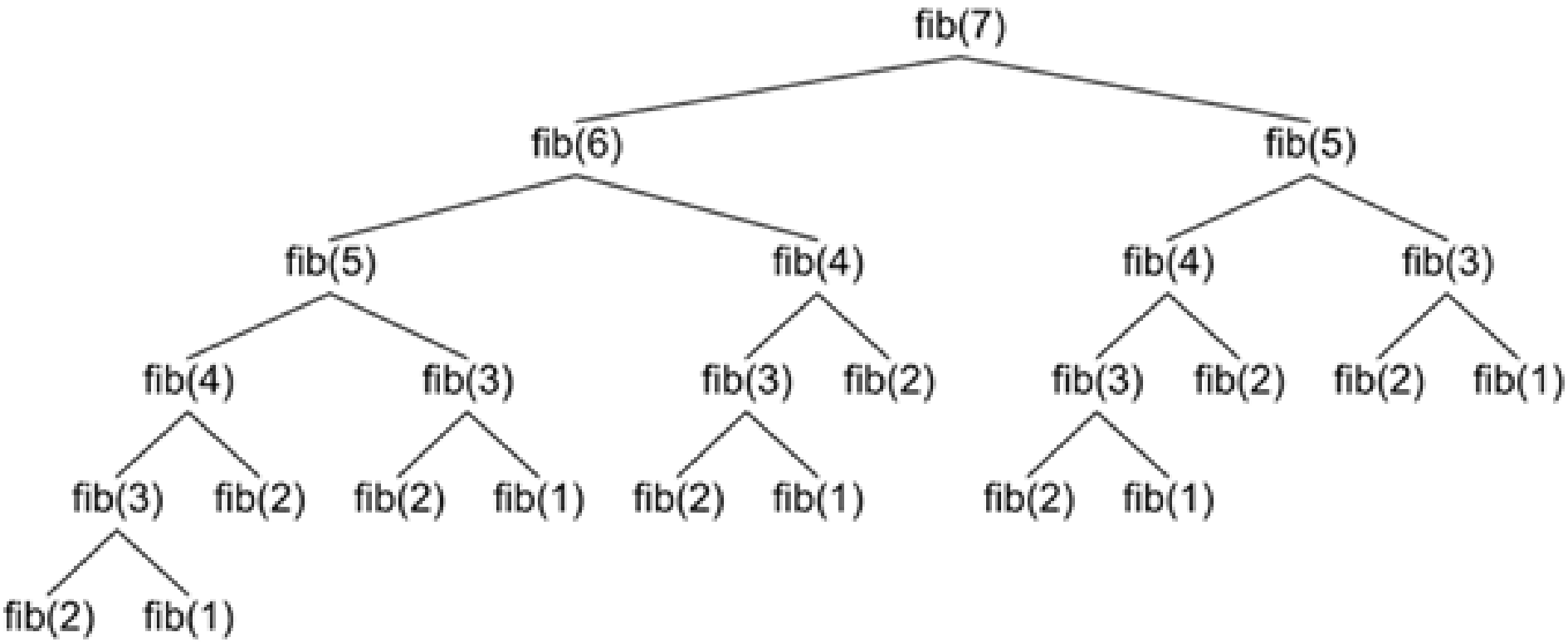
Definición

La sucesión de Fibonacci se define como $fib_0 = 0, fib_1 = 1$, $fib_n = fib_{n-1} + fib_{n-2}$ (para $n \geq 2$). Los primeros términos son $0, 1, 1, 2, 3, 5, 8, 13, 21...$

- La definición nos sugiere una forma de computarlo (función recursiva).

```
int fib(int n){
  if(n <= 1)
    return n;
  return fib(n-1) + fib(n-2);
}
```

Fibonacci (cont.)



- La complejidad es exponencial en n.

Fibonacci - dinámica

- Podemos hacerlo mejor: en lugar de recalcular la respuesta cada vez, la calculamos una sola vez y guardamos el resultado.

```
int mem[MAX_N];
int fib(int n){
  if(mem[n] >= 0)
    return mem[n]
  int res;
  if(n <= 1)
    res = n;
  else
    res = fib(n-1) + fib(n-2);
  mem[n] = res;
  return res;
}
// (al principio de main):
  fill(mem, mem+MAX_N, -1);
```

Fibonacci - dinámica

- Podemos hacerlo mejor: en lugar de recalcular la respuesta cada vez, la calculamos una sola vez y guardamos el resultado.

```
int mem[MAX_N];
int fib(int n){
  if(mem[n] >= 0)
    return mem[n]
  int res;
  if(n <= 1)
    res = n;
  else
    res = fib(n-1) + fib(n-2);
  mem[n] = res;
  return res;
}
// (al principio de main):
  fill(mem, mem+MAX_N, -1);
```

- Esto es una función recursiva
- En el arreglo mem guardamos los resultados para los n's que ya calculamos (usamos un valor negativo para indicar que no lo calculamos).
- Ahora la complejidad de calcular todos los fibonaccis hasta n es $O(n)$.

Programación dinámica - definición

- La programación dinámica es, básicamente, una estrategia donde se guardan los resultados ya calculados, sabiendo que pueden hacer falta después.
- Podemos verla como un método para resolver un problema, que lo parte en subproblemas más pequeños, los cuáles resuelve recursivamente, guardando los resultados para no calcularlos más de una vez.
- En el contexto de programación dinámica, llamamos "estado" a cada combinación de parámetros de la misma. En el caso de Fibonacci, el estado es el valor de n.
- Para calcular el costo de la programación dinámica, la cuenta suele ser "cantidad de estados" * "costo de calcular el valor de un estado"

Fibonacci - versión iterativa

```
int fib[MAXN];
void calcFibs(int n){
//  calcula todos los fibonaccis hasta n
//  y los guarda en el arreglo fib
  fib[0] = 0;
  fib[1] = 1;
  for(int i = 2; i <= n; ++i)
    fib[i] = fib[i-1] + fib[i-2];
}
```

Fibonacci - versión iterativa

```
int fib[MAXN];
void calcFibs(int n){
//  calcula todos los fibonaccis hasta n
//  y los guarda en el arreglo fib
  fib[0] = 0;
  fib[1] = 1;
  for(int i = 2; i <= n; ++i)
    fib[i] = fib[i-1] + fib[i-2];
}
```

- Todo problema de programación dinámica se puede resolver recursivamente (cómo se mostró antes) o iterativamente.
- La forma recursiva se suele llamar "memoización" (*memoization* en inglés).

Progamación dinámica iterativa vs. recursiva

- Tanto la forma iterativa como la recursiva tienen ventajas y desventajas.
- Forma iterativa:
  - Suele ser más eficiente en tiempo, porque se evitan las llamadas a función.
  - A veces permite ahorrar memoria (lo veremos más adelante).
  - El código suele resultar más corto.
- Forma recursiva:
  - Para algunos es el modo más "natural" de pensar la programación dinámica.
  - No requiere pensar en el orden en el que debemos calcular los sub-problemas, que no siempre es obvio.
  - Es más eficiente cuando hay muchos valores de la tabla que no es necesario calcular (no es muy común, pero puede ocurrir).

Números combinatorios

Definición

El número combinatorio $\binom{n}{k}$ se define como la cantidad de formas de elegir k elementos distintos de un conjunto de tamaño n.

- La forma estándar de calcular los números combinatorios es mediante la fórmula $\frac{n!}{k!(n-k)!}$.
- Si quiero calcular todos los números combinatorios de un conjunto de tamaño n, esto puede ser muy costoso. Para cada par $\binom{n}{k}$ hago O(n) cálculos, en total $O(n^3)$.
- ¿Podemos hacerlo mejor? Si. Veamos como.

Números combinatorios

Definición

El número combinatorio $\binom{n}{k}$ se define como la cantidad de formas de elegir k elementos distintos de un conjunto de tamaño n.

- Identifiquemos los elementos con los números del 1 al n.
- Si $n \geq 1$, tenemos dos opciones: tomamos el elemento n y elegimos $k-1$ elementos de los $n-1$ restantes, o no tomamos el n-ésimo y elegimos k elementos de los $n-1$ restantes.
- Luego, tenemos la recurrencia $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.
- ¡Podemos usar programación dinámica! Los casos bases serían $\binom{n}{0} = 1$, $\binom{n}{k} = 0$ para $k > n$.

Combinatorios - código recursivo

```
int mem[MAXN][MAXN];
int comb(int n, int k){
//   precondicion: mem esta inicializado en -1
  if(mem[n][k] >= 0)
    return mem[n][k];
  int res;
  if(k == 0)
    res = 1;
  else if(k > n)
    res = 0;
  else
    res = comb(n-1, k-1) + comb(n-1, k);
  mem[n][k] = res;
  return res;
}
```

Combinatorios - código iterativo

```
int comb[MAXN][MAXN];
void calcCombs(){
//  precondicion: comb esta inicializado en 0
  for(int i = 0; i < MAXN; ++i){
      comb[i][0] = comb[i][i] = 1;
        for(int j = 1; j < i; ++j)
          comb[i][j] = comb[i-1][j-1] + comb[i-1][j];
    }
}
```

- Esta tabla se conoce comúnmente como "Triángulo de Pascal"
- La complejidad de ambas versiones es $O(MAXN^2)$
- Los números combinatorios aparecen en muchos problemas de conteo y probabilidad. Es útil saber implementar rápido la versión iterativa de la tabla.

Problemas de optimización

- La programación dinámica se puede usar también para resolver algunos problemas de optimización (encontrar la mejor solución, de acuerdo a cierto criterio).
- ¿Se acuerdan del problema de la moneda? (el que vimos en su versión greedy). En el caso general se puede resolver con programación dinámica.

Problema de la moneda

Enunciado

Tenemos n tipos de monedas de valores $v_0, v_1, ..., v_{n-1}$.
Queremos saber la menor cantidad de monedas que necesitamos para sumar exactamente m.

- Definimos $f(k, j) =$ "menor cantidad de monedas que necesitamos para sumar exactamente j usando sólo los primeros k valores de monedas $(v_0, ..., v_{k-1})$". Nos interesa calcular $f(n, m)$.
- Si $k \geq 1$ consideramos dos opciones:
  - No usamos la moneda de valor v_{k-1}. Esto es lo mismo que resolver el subproblema para $(k-1, j)$.
  - Usamos la moneda de valor v_{k-1} al menos una vez. Esto es lo mismo que resolver el subproblema para $(k, j - v_{k-1})$ y agregar una moneda de valor v_{k-1}.
- Esto nos da la recurrencia:
  $f(k, j) = min(f(k-1, j), 1 + f(k, j - v_{k-1}))$.

Problema de la moneda - recurrencia

Agregando los casos bases, queda:

$$f(k,j) = \begin{cases} 0, & \text{si } k = 0 \text{ y } j = 0 \\ \infty, & \text{si } (k = 0 \text{ y } j \neq 0) \text{ ó } j < 0 \\ min(f(k-1,j), 1 + f(k, j - v_{k-1})), & \text{caso contrario} \end{cases}$$

- Como lo que queremos es un mínimo, el valor ∞ representa la ausencia de solución. A nivel código, se puede usar un valor lo suficientemente alto, de modo que nunca una solución pueda tener ese tamaño.
- La complejidad es $O(n * m)$

Problema de la moneda - Código recursivo

```
int v[MAXN];
int mem[MAXN][MAXM];
int f(int k, int j){
//   precondicion: mem esta inicializado en -1
   if(j < 0)
      return INF;
   if(mem[k][j] >= 0)
      return mem[n][k];
   int res;
   if(k == 0){
      if(j == 0)
         res = 0;
      else
         res = INF;
   }
   else
      res = min(f(k-1,j),1+f(k,j-v[k-1]));
   mem[k][j] = res;
   return res;
}
```

Problema de la moneda - Código iterativo

```
int v[MAXN];
int f[MAXN][MAXM];
int solve(int n, int m){
  f[0][0] = 0;
  for(int j = 1; j <= m; ++j)
    f[0][j] = INF;
  for(int k = 1; k <= n; ++k){
    for(int j = 0; j <= m; ++j){
      f[k][j] = f[k-1][j];
      if(v[k-1] <= j)
        f[k][j]=min(f[k][j],1+f[k][j-v[k-1]]);
    }
  }
  return f[n][m];
}
```

Moneda - Generar solución

- En muchos problemas de optimización, además del valor de la solución óptima, se nos pide que demos explícitamente la solución (en el caso de moneda, cuántas monedas tengo que tomar de cada tipo).
- Podemos aprovechar que tenemos guardado el resultado de todos los sub-problemas.
- Empezamos desde el estado (n, m)
  - Si $f(n, m) = f(n-1, m)$ recursionamos para $(n-1, m)$.
  - Si no, es porque tenemos que agregar la moneda $n-1$. La agregamos al resultado y recursionamos para $(n, m - v_{n-1})$.
  - Seguimos hasta llegar a $(0, 0)$.

Mondeda - Código para generar solución

```
int f[MAXN][MAXM];
int sol[MAXN];
void gen_rec(int k, int j){
  if(k == 0)
    return;
  if(f[k][j] == f[k-1][j])
    gen_rec(k-1, j);
  else {
    sol[k-1]++;
    gen_rec(k, j - v[k][j-v[k-1]]);
  }
}
void generate(int n, int m){
//  guarda en sol cuantas monedas necesito para cada valor
//  precondicion: f contiene los resultados para cada
    subproblema
  fill(sol, sol+n, 0);
  gen_rec(n,m)
}
```

Generar solución

- Este patrón para generar una solución óptima se puede aplicar a cualquier programación dinámica que resuelva un problema de optimización.
- Partimos del estado que nos interesa.
- En cada paso usamos los valores calculados para ver qué decisión coincide con el resultado del estado.
- Guardamos en la solución la decisión que tomamos (en el caso de moneda, incrementamos la cantidad de monedas del valor que consideramos en el caso de que nos conviene).
- Recursionamos para los sub-problemas.

Moneda - optimización de memoria

- Para muchos problemas podemos ahorrar memoria haciéndolo de forma iterativa.
- Hay que prestar atención a cuando se puede y cuando es necesario.

```
int v[MAXN], f[MAXM];
int solve(int n, int m){
  f[0] = 0;
  fill(f+1, f+m+1, INF);
  for(int k = 0; k < n; ++k)
    for(int j = v[k]; j <= m; ++j)
      f[j] = min(f[j], 1 + f[j-v[k]]);
  return f[m];
}
```

Moneda - optimización de memoria

- Desventaja: no nos permite reconstruir la solución.
- Ventaja: Si bien la complejidad es la misma, tener menos memoria reduce el tiempo de ejecución porque disminuye los fallos de caché.
- Algunos problemas requieren esta optimización para entrar en los límites de tiempo y memoria (pricipalmente memoria).

Sumas parciales - 1D

Enunciado

Dado un arreglo de números $a_0 ... a_{n-1}$. Definimos $sum(l, r) =$ "suma del sub-arreglo $a_l ... a_{r-1}$". Dadas q pares (l_i, r_i), queremos calcular $sum(l_i, r_i)$ para cada uno.

- La solución más obvia es recorrer el sub-arreglo correspondiente para cada pregunta, calculando la suma en $O(n)$.
- Podemos hacerlo mejor observando que $sum(l, r) = sum(0, r) - sum(0, l)$
- Luego, si precalculamos $sum(0, i)$ para todo i, podemos responder cada pregunta en $O(1)$.
- La recurrencia es simple:
  - $sum(0, 0) = 0$
  - $sum(0, i + 1) = sum(0, i) + a_i$.

Sumas parciales - 1D - Código

```
int a[MAXN],sp[MAXN+1];int n;
void init_sum(){
  sp[0] = 0;
  for(int i = 0; i < n; ++i)
    sp[i+1] = sp[i] + a[i];
}
int sum(int l, int r){
  return sp[r] - sp[l];
}
```

La idea de hacer sumas parciales aparece en muchos problemas, es bueno tenerla presente.

Sumas parciales - 2D

Enunciado

Ahora, queremos resolver el mismo problema, pero en dos dimensiones. Es decir, en vez de tener un arreglo tenemos una matriz, y en vez de calcular la suma de un intervalo queremos calcular la suma de un "rectángulo" de la matriz. Concretamente, tenemos una matriz $a_{i,j}$ ($i \in [0, n), j \in [0, m)$) y queremos calcular

$$sum(i_0, j_0, i_1, j_1) = \sum_{i=i_0}^{i_1-1} \sum_{j=j_0}^{j_1-1} a_{i,j}$$

- Dibujito en pizarrón
- Podemos aplicar una idea similar al caso de una dimensión.

Sumas parciales - 2D - cont.

- Definimos $sp(i, j) = sum(0, 0, i, j)$ (suma del rectángulo entre $(0, 0)$ y $(i - 1, j - 1)$).
- Tenemos que
  $sum(i_0, j_0, i_1, j_1) = sp(i_1, j_1) - sp(i_1, j_0) - sp(i_0, j_1) + sp(i_0, j_0)$.
  (dibujito en pizarrón).
- Entonces, podemos responder en $O(1)$ si precalculamos $sp(i, j)$.
- Tenemos que (para $i > 0, j > 0$)
  $sp(i, j) = sp(i, j - 1) + sp(i - 1, j) + a_{i-1,j-1} - sp(i - 1, j - 1)$.
  (dibujito en pizarrón).
- Podemos usar programación dinámica para calcular esta recurrencia.
- Código: ejercicio.

Edit distance

http://www.spoj.com/problems/EDIST/

Enunciado

Dados dos strings A y B, decir la menor cantidad de operaciones necesarias para transformar A en B. Las operaciones permitidas son:

- Agregar un caracter en cualquier posición.
- Borrar un caracter.
- Cambiar un caracter.

Por ejemplo, si $A =$ "saturday", $B =$ "sundays" una solución es: "saturday" $\rightarrow$ "sturday" $\rightarrow$ "surday" $\rightarrow$ "sunday" $\rightarrow$ "sundays".

- Llamemos $f(A, B)$ al resultado.
- Los casos en que A ó B son vacíos son fáciles ($f(\text{""}, B) = |B|$, $f(A, \text{""}) = |A|$).

Edit distance (cont.)

- Si A y B no son vacíos, entonces $A = A'a$, $B = B'b$ (llamamos a y b al último caracter de A y B respectivamente).
- Si $a = b$, es claro que $f(A, B) = f(A', B')$.
- Si $a \neq b$, entonces tenemos que realizar operaciones sobre A para que su último caracter sea b. Tenemos varias opciones:
  - Agregamos el caracter b al final de A y resolvemos para (A, B')
  - Borramos el último caracter a y resolvemos para (A', B).
  - Cambiamos el último caracter de A por b y resolvemos para (A', B').
- Es fácil convencerse de que este método contempla todas las posibilidades óptimas.
- Con estas observaciones, se puede armar una recurrencia sobre (i, j) que calcule "menor costo de transformar los primeros i caracteres de A en los primeros j caracteres de B" .
- La recurrencia y el código quedan como ejercicio.

Conclusión

- Programación dinámica es uno de los tópicos más comunes en el contexto de programación competitiva, por lo que es muy importante dedicarle tiempo a practicarlo.
- Si lo permite el cronograma, en una clase posterior veremos cosas más avanzadas acerca de programación dinámica (algunos patrones que suelen aparecer, técnicas de optimización).
- Un concepto muy relacionado a programación dinámica del cual no hablamos es "backtracking". Es lo mismo que programación dinámica, pero sin guardar información de los sub-problemas resueltos (sólo la recursión que prueba todas las posibilidades). Se usa backtracking cuando la estructura del problema es tal que cada sub-problema se resuelve una sóla vez.