Programación Dinámica

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2025

Contenidos

Gracias Sponsors!

Organizador

Diamond Plus

Gracias Sponsors!

Platino

FOLDER IT

INTERNATIONAL
SOFTWARE COMPANY

Gold

NeuralSoft

Oro

JS JERÁRQUICOS

Gracias Sponsors!

Aliado

“Pay heed to the tales of old wives. It may well be that they alone keep in memory what it was once needful for the wise to know.”

J. R. R. Tolkien, The Lord of the Rings

“Memory is the thing you forget with.”

Alexander Chase, Perspectives, 1966.

“I cannot but remember such things were,
That were most precious to me.”

William Shakespeare, Macbeth
Act IV, scene 3, line 222

Contenidos

Sucesión de Fibonacci

Sucesión de Fibonacci

La sucesión de Fibonacci se define como $f_0 = 1$, $f_1 = 1$ y $f_{n+2} = f_n + f_{n+1}$ para todo $n \geq 0$

Sucesión de Fibonacci

Sucesión de Fibonacci

La sucesión de Fibonacci se define como $f_0 = 1$, $f_1 = 1$ y $f_{n+2} = f_n + f_{n+1}$ para todo $n \geq 0$

¿Cómo podemos computar el término 100 de la sucesión de Fibonacci?

Sucesión de Fibonacci

Sucesión de Fibonacci

La sucesión de Fibonacci se define como $f_0 = 1$, $f_1 = 1$ y $f_{n+2} = f_n + f_{n+1}$ para todo $n \geq 0$

¿Cómo podemos computar el término 100 de la sucesión de Fibonacci?
¿Cómo podemos hacerlo eficientemente?

Algoritmos recursivos

Definición

Recursivo: calcula *instancias* de X en función de *otras instancias* de X, hasta llegar a un *caso base* cuya respuesta se puede calcular directamente sin necesitar otras instancias de X.

Algoritmos recursivos

Definición

Recursivo: calcula *instancias* de X en función de *otras instancias* de X, hasta llegar a un *caso base* cuya respuesta se puede calcular directamente sin necesitar otras instancias de X.

Ejemplo

Para calcular el factorial de un número, un posible algoritmo es calcular Factorial(n) como Factorial($n-1$) $\times n$ si $n \geq 1$ o 1 si $n = 0$

Algoritmos recursivos

Definición

Recursivo: calcula *instancias* de X en función de *otras instancias* de X, hasta llegar a un *caso base* cuya respuesta se puede calcular directamente sin necesitar otras instancias de X.

Ejemplo

Para calcular el factorial de un número, un posible algoritmo es calcular Factorial(n) como Factorial($n-1$) $\times n$ si $n \geq 1$ o 1 si $n = 0$

Veamos como calcular el n-ésimo fibonacci con un algoritmo recursivo

Cálculo Recursivo de Fibonacci

```
1  int fibo(int n)
2  {
3      if(n<=1)
4          return 1;
5      else
6          return fibo(n-2)+fibo(n
               -1);
7  }
```

Cálculo Recursivo de Fibonacci

```
1 | int fibo(int n)
2 | {
3 |     if(n<=1)
4 |         return 1;
5 |     else
6 |         return fibo(n-2)+fibo(n
  |                -1);
7 | }
```

Notemos que fibo(n) llama a fibo($n-2$), pero después vuelve a llamar a fibo($n-2$) para calcular fibo($n-1$), y a medida que va decreciendo el parámetro que toma fibo son más las veces que se llama a la función fibo con ese parámetro.

Problemas de la recursión

- La función tiene un problema.

Problemas de la recursión

- La función tiene un problema.
- Llamamos muchas veces con los mismos parámetros

Problemas de la recursión

- La función tiene un problema.
- Llamamos muchas veces con los mismos parámetros
- Básicamente tenemos “problemas de memoria”.

Problemas de la recursión

- La función tiene un problema.
- Llamamos muchas veces con los mismos parámetros
- Básicamente tenemos “problemas de memoria”.
- ¿Podemos solucionar esto?

Problemas de la recursión

- La función tiene un problema.
- Llamamos muchas veces con los mismos parámetros
- Básicamente tenemos “problemas de memoria”.
- ¿Podemos solucionar esto?
- ¡Sí! Eso es programación dinámica.

Contenidos

Programación dinámica

But do not despise the lore that has come down from distant years; for oft it may chance that old wives keep in memory word of things that once were needful for the wise to know.

Celeborn the Wise,
The Fellowship of the Ring,
J. R. R. Tolkien

- La programación dinámica consiste, esencialmente, en una recursión con una suerte de memorización o cache.

Programación dinámica

Visión tradicional:

La programación dinámica es una técnica que consiste en:

Programación dinámica

Visión tradicional:

La programación dinámica es una técnica que consiste en:

- Dividir un problema en subproblemas, o reducirlo a instancias más fáciles de calcular del problema.

Programación dinámica

Visión tradicional:

La programación dinámica es una técnica que consiste en:

- Dividir un problema en subproblemas, o reducirlo a instancias más fáciles de calcular del problema.
- Resolver las instancias de cada subproblema de la misma manera (dividiendo en subproblemas o reduciendo a otra instancia) hasta llegar a un caso base.

Programación dinámica

Visión tradicional:

La programación dinámica es una técnica que consiste en:

- Dividir un problema en subproblemas, o reducirlo a instancias más fáciles de calcular del problema.
- Resolver las instancias de cada subproblema de la misma manera (dividiendo en subproblemas o reduciendo a otra instancia) hasta llegar a un caso base.
- Guardar el resultado de cada instancia del problema la primera vez que se calcula, para que cada vez que se vuelva a necesitar el valor de esa instancia ya esté almacenado, y no sea necesario calcularlo nuevamente.

Programación dinámica

Visión tradicional:

La programación dinámica es una técnica que consiste en:

- Dividir un problema en subproblemas, o reducirlo a instancias más fáciles de calcular del problema.
- Resolver las instancias de cada subproblema de la misma manera (dividiendo en subproblemas o reduciendo a otra instancia) hasta llegar a un caso base.
- Guardar el resultado de cada instancia del problema la primera vez que se calcula, para que cada vez que se vuelva a necesitar el valor de esa instancia ya esté almacenado, y no sea necesario calcularlo nuevamente.

¿Cómo hacemos para calcular una sóla vez una función para cada parámetro, por ejemplo, en el caso de Fibonacci?

Cálculo de Fibonacci mediante Programación Dinámica

```
1  int fibo[100];
2  int calcFibo(int n)
3  {
4      if(fibo[n]!=-1)
5          return fibo[n];
6      fibo[n] = calcFibo(n-2)+calcFibo(n-1);
7      return fibo[n];
8  }
9  int main()
10 {
11     for(int i=0;i<100;i++)
12         fibo[i] = -1;
13     fibo[0] = 1;
14     fibo[1] = 1;
15     int fibo50 = calcFibo(50);
16 }
```

Ventajas de la Programación Dinámica

- Comparada con la primera versión recursiva que vimos, programación dinámica llama menos veces a cada función

Ventajas de la Programación Dinámica

- Comparada con la primera versión recursiva que vimos, programación dinámica llama menos veces a cada función
- calcFibo(n-1) necesita calcular calcFibo(n-2), pero ya lo calculamos antes, por lo que no es necesario volver a llamar a calcFibo(n-3) y calcFibo(n-4)

Ventajas de la Programación Dinámica

- Comparada con la primera versión recursiva que vimos, programación dinámica llama menos veces a cada función
- calcFibo(n-1) necesita calcular calcFibo(n-2), pero ya lo calculamos antes, por lo que no es necesario volver a llamar a calcFibo(n-3) y calcFibo(n-4)
- Este algoritmo es lineal. El anterior era exponencial.
- Esta forma particularmente sencilla de implementar programación dinámica, es decir, mediante el agregado al código exacto de la recursión ingenua, un par de líneas para guardar y leer respuestas previas de la cache, se llama *memoización* (del inglés *memoization*).

Números combinatorios

Ejemplo

Otro ejemplo de un problema que puede ser resuelto mediante programación dinámica es el de los números combinatorios

Números combinatorios

Ejemplo

Otro ejemplo de un problema que puede ser resuelto mediante programación dinámica es el de los números combinatorios

Cómo lo calculamos

El combinatorio $\binom{n}{k}$ se puede calcular como $\frac{n!}{k!(n-k)!}$, pero si en lugar de un único combinatorio queremos precalcular una tabla completa de combinatorios, lo más práctico es usar el procedimiento del triángulo de pascal.

Calculo recursivo del número combinatorio

Algoritmo recursivo

```
1  int comb(int n, int k)
2  {
3      if(k==0||k==n)
4          return 1;
5      else
6          return comb(n-1,k-1)+comb(n-1,k);
7  }
```

Calculo recursivo del número combinatorio

Algoritmo recursivo

```
1  int comb(int n, int k)
2  {
3      if(k==0||k==n)
4          return 1;
5      else
6          return comb(n-1,k-1)+comb(n-1,k);
7  }
```

- Este algoritmo tiene un problema. ¿Cuál es el problema?

Calculo recursivo del número combinatorio

Algoritmo recursivo

```
1  int comb(int n, int k)
2  {
3      if(k==0||k==n)
4          return 1;
5      else
6          return comb(n-1,k-1)+comb(n-1,k);
7  }
```

- Este algoritmo tiene un problema. ¿Cuál es el problema?
- Calcula muchas veces el mismo número combinatorio. ¿Cómo arreglamos esto?

Número combinatorio calculado con programación dinámica

Algoritmo con Programación Dinámica

```
1  int comb[100][100];
2  void llenarTablaCombinatoriosHasta(int n)
3  {
4      for (int i = 0; i <= n; i++)
5      {
6          comb[i][0] = comb[i][i] = 1;
7          for (int k = 1; k < i; k++)
8              comb[i][k] = comb[i-1][k] + comb[i-1][k-1];
9      }
10 }
```

Número combinatorio calculado con programación dinámica (continuado)

- No utiliza memoización: forma *bottom-up*.
- Esta forma es mucho más eficiente que memoización, **cuando se calculan muchas entradas** de la tabla cache.

Superposición de subproblemas

- El beneficio de la programación dinámica se da justamente porque evitamos recalcular subproblemas que se superponen.
- Sin superposición de subproblemas, podríamos usar programación dinámica, pero la complejidad extra de la cache no aportaría nada, porque nunca se reutilizaría un subproblema ya calculado, y podríamos haber usado directamente una recursión común (divide and conquer).

Contenidos

Otro tipo de problemas

- Hasta ahora vimos problemas en los que hay que calcular una *cantidad* determinada.

Otro tipo de problemas

- Hasta ahora vimos problemas en los que hay que calcular una *cantidad* determinada.
- En algunos casos lo que tenemos que calcular es el mínimo o máximo de alguna función objetivo, sobre un conjunto de soluciones posibles, es decir, resolver un problema de *optimización*.

Otro tipo de problemas

- Hasta ahora vimos problemas en los que hay que calcular una *cantidad* determinada.
- En algunos casos lo que tenemos que calcular es el mínimo o máximo de alguna función objetivo, sobre un conjunto de soluciones posibles, es decir, resolver un problema de *optimización*.
- En estos casos para poder usar programación dinámica necesitamos asegurarnos de que la solución de los subproblemas sea parte de la solución de la instancia original del problema.

Principio de optimalidad

Definición

Se dice que un problema cumple el *principio de optimalidad de Bellman* cuando en una o más partes de una solución óptima al mismo, debe aparecer necesariamente una solución óptima a un subproblema.

Principio de optimalidad

Definición

Se dice que un problema cumple el *principio de optimalidad de Bellman* cuando en una o más partes de una solución óptima al mismo, debe aparecer necesariamente una solución óptima a un subproblema.

- Bellman, quien estudió la programación dinámica en 1953, afirmó que el principio de optimalidad es un requisito indispensable para poder aplicar esta técnica.
- Notar que justamente es el principio de optimalidad lo que nos permite utilizar recursión.

Principio de optimalidad

Definición

Se dice que un problema cumple el *principio de optimalidad de Bellman* cuando en una o más partes de una solución óptima al mismo, debe aparecer necesariamente una solución óptima a un subproblema.

- Bellman, quien estudió la programación dinámica en 1953, afirmó que el principio de optimalidad es un requisito indispensable para poder aplicar esta técnica.
- Notar que justamente es el principio de optimalidad lo que nos permite utilizar recursión.

Veamos un ejemplo de problema clásico.

Viaje óptimo en matriz

- Supongamos que tenemos una matriz de $n \times m$, con números enteros, y queremos encontrar el camino de la esquina **superior-izquierda**, a la esquina **inferior-derecha**, que maximice la suma de las casillas recorridas.
- El camino está restringido a utilizar únicamente movimientos de tipo **derecha** y **abajo**.
- ¿Cómo resolveríamos este problema?

Viaje óptimo en matriz

- Supongamos que tenemos una matriz de $n \times m$, con números enteros, y queremos encontrar el camino de la esquina **superior-izquierda**, a la esquina **inferior-derecha**, que maximice la suma de las casillas recorridas.
- El camino está restringido a utilizar únicamente movimientos de tipo **derecha** y **abajo**.
- ¿Cómo resolveríamos este problema?
- $f(x, y) = M_{x,y} + max(f(x-1, y), f(x, y-1))$
- $f(1, y) = M_{1,y} + f(1, y-1))$
- $f(x, 1) = M_{x,1} + f(x-1, 1))$
- $f(1, 1) = M_{1,1}$

Viaje óptimo en matriz (camino)

- La recursión anterior lleva a un algoritmo de programación dinámica para calcular la suma óptima en el recorrido.
- ¿Pero cómo recuperaríamos el camino en sí?

Viaje óptimo en matriz (camino)

- La recursión anterior lleva a un algoritmo de programación dinámica para calcular la suma óptima en el recorrido.
- ¿Pero cómo recuperaríamos el camino en sí?
- ¿Y si quisiéramos el camino lexicográficamente mínimo?

Viaje óptimo en matriz (camino)

- La recursión anterior lleva a un algoritmo de programación dinámica para calcular la suma óptima en el recorrido.
- ¿Pero cómo recuperaríamos el camino en sí?
- ¿Y si quisiéramos el camino lexicográficamente mínimo?
- ¿Y si quisiéramos el camino número 420 en orden lexicográfico?

Viaje óptimo en matriz (camino)

- La recursión anterior lleva a un algoritmo de programación dinámica para calcular la suma óptima en el recorrido.
- ¿Pero cómo recuperaríamos el camino en sí?
- ¿Y si quisiéramos el camino lexicográficamente mínimo?
- ¿Y si quisiéramos el camino número 420 en orden lexicográfico?

Contenidos

Visión constructiva de DP

- En el ejemplo anterior, nos encontramos con que definimos la recursión de una cierta manera, pero después nos quedó **"al revés"**.
- Notar que si solamente quisiéramos calcular el resultado (un único numerito), daría lo mismo el orden o la recursión concreta.
- El problema surge al querer reconstruir el camino:
  - Pensamos un **procedimiento de construcción de la solución**
  - Ese procedimiento está formado por una secuencia ordenada de **pasos**, que son **parte del problema** y queremos poder reconstruir
  - Cada paso transforma un **estado** intermedio (o subproblema) en otro, durante el proceso de construcción de la solución.
  - ¡Pero nada de esto fue tenido en cuenta cuando pensamos la construcción!
- Por lo tanto, si nos importa reconstruir o razonar sobre el camino, se puede pensar programación dinámica de otra forma.

Visión constructiva de DP (continuado)

- Identificamos un proceso de construcción de la solución, identificando **estados** y **transiciones**.
- El estado debe contener toda la **información** relevante que necesitaremos para poder decidir las transiciones óptimas.
- Las transiciones posibles no deben producir ciclos, (o nuestra recursión se volvería infinita).
- Ahora cada estado se corresponde a una entrada en la tabla o cache de nuestro algoritmo de DP, y las transiciones vienen dadas por las llamadas recursivas.

Visión constructiva de DP (continuado)

- Si aplicamos esto al problema anterior, la función y estados nos quedan al revés que antes: Esto es “de atrás para adelante”, pero notar que nos permite reconstruir el camino en el orden natural.
- ¡Ahora sí podemos encontrar el camino lexicográficamente mínimo!
- También podemos encontrar si utilizamos este orden, el camino 420 en orden lexicográfico. Ejercicio para pensar.

Contenidos

Ejemplos

- Mismo problema de antes, pero ahora podemos hacer hasta K movimientos del tipo **izquierda**.

Ejemplos

- Mismo problema de antes, pero ahora podemos hacer hasta K movimientos del tipo **izquierda**.
- Mismo problema de arriba, pero además podemos usar hasta W elefantes mágicos, que nos permiten hacer un movimiento de alfil de ajedrez en la matriz.

Ejemplos

- Mismo problema de antes, pero ahora podemos hacer hasta K movimientos del tipo **izquierda**.
- Mismo problema de arriba, pero además podemos usar hasta W elefantes mágicos, que nos permiten hacer un movimiento de alfil de ajedrez en la matriz.
- Mismo problema de arriba, pero además podemos usar hasta Z globos aerostáticos, que nos permiten hacer un movimiento de torre de ajedrez en la matriz.

Ejemplos (más)

- Subset sum: Dados n números enteros positivos, decidir si se puede obtener una suma S usando algunos de ellos (a lo sumo una vez cada uno).
- Knapsack o problema de la mochila: Dados objetos con peso y valor, queremos meter el máximo valor posible en una mochila que soporta hasta P de peso.
- Resolvamos ambos pensándolos de la manera constructiva (¡Por ejemplo, porque nos piden la solución lexicográficamente más chica, y no queremos tener que reescribir todo al final porque nos quedó al revés!)

Resumiendo

- Hemos visto dos maneras de pensar la programación dinámica:
  - Tradicional (recursiva)
  - Pensando en un proceso de construcción de solución (constructiva)
- Si solo interesa el numerito, usar cualquier opción de las anteriores (la que resulte más natural o fácil).
- Si interesa reconstruir el camino, y muy especialmente si interesa algún camino en orden lexicográfico, pensarlo de la manera constructiva puede ser útil.

Resumiendo (continuado)

- Y dos maneras de implementarla:
  - Memoización (La más fácil, no hay que pensar en qué orden iterar los subproblemas. Buena cuando solo se usan algunos pocos subproblemas, difíciles de caracterizar o iterar)
  - Bottom-up (Más eficiente si se usan casi todos los estados posibles, pero requiere poder iterar todos los estados relevantes, y pensar con cuidado el orden de recorrida)

Contenidos

Introducción

- Patrón “Dinámicas en rangos”: típico. Aparece mucho.
- Subproblemas/estados de la forma (a, b): intervalos o rangos.
- Lo mejor es estudiar algunos ejemplos concretos.

Contenidos

ABB óptimo

- Dada una secuencia de valores ordenados $v_1 < v_2 < \cdots < v_n$, junto a sus frecuencias o probabilidades de aparición $f_1, \cdots, f_n$, dar un algoritmo que compute un árbol binario de búsqueda que minimice el tiempo (cantidad de comparaciones realizadas = profundidad) esperado para encontrar un elemento.

Paréntesis

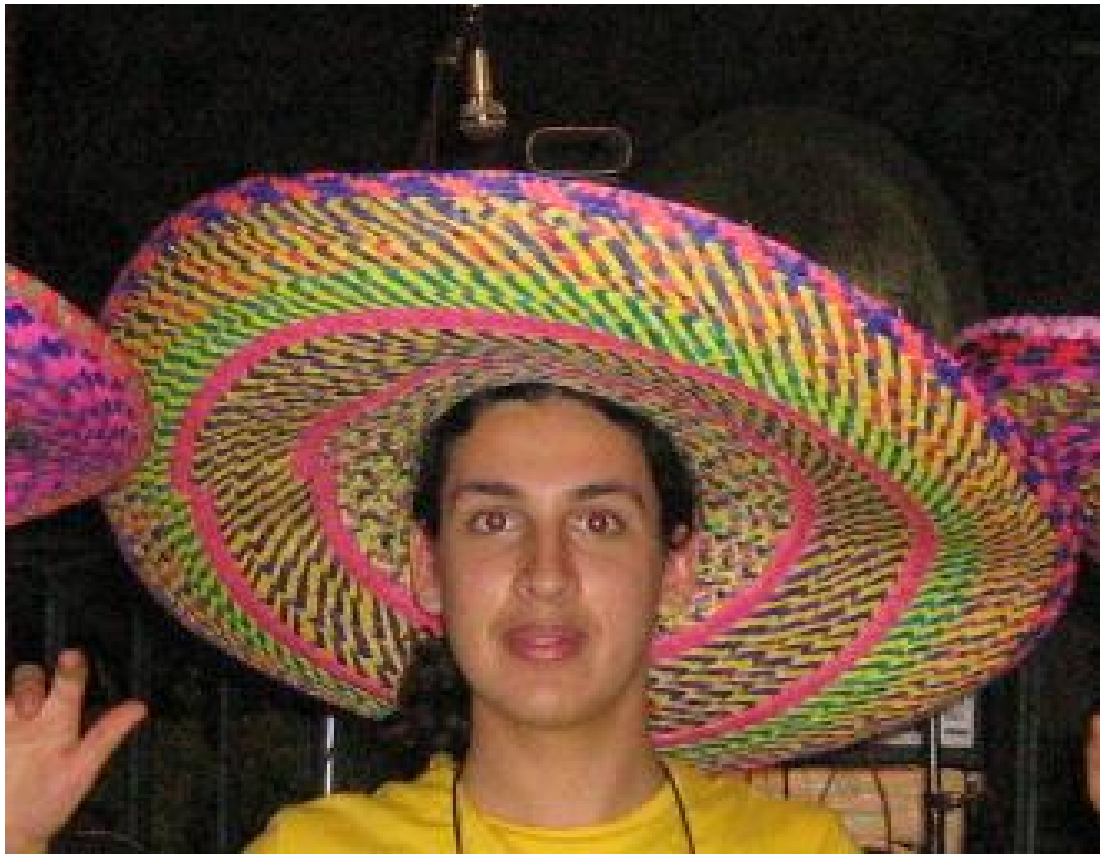
- Dada una cadena de caracteres {, }, [,] , (y), de longitud par, dar la mínima cantidad de reemplazos de caracteres que se le deben realizar a este string para dejar una secuencia “bien parenteseada”.
- Las secuencias bien parenteseadas T son las que se pueden formar con las siguientes reglas, a partir de secuencias bien parenteseadas anteriormente obtenidas S y S':
  - $T = \emptyset$
  - $T = SS'$
  - $T = (S)$
  - $T = [S]$
  - $T = \{S\}$

Contenidos

Tarea

Problema de la IOI de México 2006

- `https://wcipeg.com/problem/ioi0621`



Problemas de práctica

- https://cses.fi/problemset/task/1633
- https://cses.fi/problemset/task/1637
- https://cses.fi/problemset/task/1634
- https://cses.fi/problemset/task/1635
- https://cses.fi/problemset/task/1745
- https://cses.fi/problemset/task/1638
- https://cses.fi/problemset/task/1158
- https://cses.fi/problemset/task/3403
- https://cses.fi/problemset/task/1744
- http://goo.gl/BJtPZ
- http://codeforces.com/problemset/problem/225/C
- http://codeforces.com/problemset/problem/163/A