Programación Dinámica

Leopoldo Taravilse

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2012

Contenidos

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

Un algoritmo se dice recursivo si calcula instancias de un problema en función de otras instancias del mismo problema hasta llegar a un caso base, que suele ser una instancia pequeña del problema, cuya respuesta generalmente está dada en el algoritmo y no es necesario calcularla.

Algoritmos recursivos

Definición

Un algoritmo se dice recursivo si calcula instancias de un problema en función de otras instancias del mismo problema hasta llegar a un caso base, que suele ser una instancia pequeña del problema, cuya respuesta generalmente está dada en el algoritmo y no es necesario calcularla.

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

Un algoritmo se dice recursivo si calcula instancias de un problema en función de otras instancias del mismo problema hasta llegar a un caso base, que suele ser una instancia pequeña del problema, cuya respuesta generalmente está dada en el algoritmo y no es necesario calcularla.

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 algorimto 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 que usamos para calcular Fibonacci tiene un problema.

Problemas de la recursión

- La función que usamos para calcular Fibonacci tiene un problema.
- Llamamos muchas veces a la misma función con los mismos parámetros

Problemas de la recursión

- La función que usamos para calcular Fibonacci tiene un problema.
- Llamamos muchas veces a la misma función con los mismos parámetros
- ¿Podemos solucionar esto? ¿Podemos hacer que la función sea llamada pocas veces para cada parámetro?

Problemas de la recursión

- La función que usamos para calcular Fibonacci tiene un problema.
- Llamamos muchas veces a la misma función con los mismos parámetros
- ¿Podemos solucionar esto? ¿Podemos hacer que la función sea llamada pocas veces para cada parámetro?
- Para lograr resolver este problema, vamos a introducir el concepto de programación dinámica

Contenidos

Programación dinámica

La programación dinámica es una técnica que consiste en:

Programación dinámica

La programación dinámica es una técnica que consiste en:

- Dividir un problema en dos o más subproblemas o reducirlo a una instancia más fácil de calcular del problema.

Programación dinámica

La programación dinámica es una técnica que consiste en:

- Dividir un problema en dos o más subproblemas o reducirlo a una instancia más fácil 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

La programación dinámica es una técnica que consiste en:

- Dividir un problema en dos o más subproblemas o reducirlo a una instancia más fácil 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

La programación dinámica es una técnica que consiste en:

- Dividir un problema en dos o más subproblemas o reducirlo a una instancia más fácil 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

- La función que vimos recién que usa programación dinámica tiene una ventaja con respecto a la versión recursiva que vimos anteriormente.

Ventajas de la Programación Dinámica

- La función que vimos recién que usa programación dinámica tiene una ventaja con respecto a la versión recursiva que vimos anteriormente.
- Llama menos veces a cada función

Ventajas de la Programación Dinámica

- La función que vimos recién que usa programación dinámica tiene una ventaja con respecto a la versión recursiva que vimos anteriormente.
- Llama menos veces a cada función
- Para calcular 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

- La función que vimos recién que usa programación dinámica tiene una ventaja con respecto a la versión recursiva que vimos anteriormente.
- Llama menos veces a cada función
- Para calcular 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)
- Así podemos calcular calcFibo(50) mucho más rápido ya que este algoritmo es lineal mientras que el anterior era exponencial.

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 generalmente como es un número muy grande, se suele tener que calcular módulo P, para algún entero P que suele ser un primo bastante grande. Dividir módulo P involucra hacer muchas cuentas y no tan sencillas, por lo que lo más eficiente es calcular $\binom{n}{k}$ como $\binom{n-1}{k-1} + \binom{n-1}{k}$, salvo que k sea 0 o n en cuyo caso el número combinatorio es 1.

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  int calcComb(int n, int k)
3  {
4      if(comb[n][k]!=-1)
5          return comb[n][k];
6      if(k==0||k==n)
7          comb[n][k] = 1;
8      else
9          comb[n][k] = calcComb(n-1,k-1)+calcComb(n-1,k);
10     return comb[n][k];
11 }
```

Número combinatorio calculado con programación dinámica

Algoritmo con Programación Dinámica

```
1  int comb[100][100];
2  int calcComb(int n, int k)
3  {
4      if(comb[n][k]!=-1)
5          return comb[n][k];
6      if(k==0||k==n)
7          comb[n][k] = 1;
8      else
9          comb[n][k] = calcComb(n-1,k-1)+calcComb(n-1,k);
10     return comb[n][k];
11 }
```

Este algoritmo asume que comb está inicializado en -1 en todas sus posiciones.

Problemas

Algunos problemas para practicar programación dinámica.

- https://www.spoj.pl/problems/BORW/
- http://goo.gl/qP6p8

Contenidos

Otro tipo de problemas

- Hasta ahora vimos problemas en los que hay que calcular un número de terminado.

Otro tipo de problemas

- Hasta ahora vimos problemas en los que hay que calcular un número de terminado.
- En algunos casos lo que tenemos que calcular es el menor o el mayor número que cumple con cierta propiedad.

Otro tipo de problemas

- Hasta ahora vimos problemas en los que hay que calcular un número de terminado.
- En algunos casos lo que tenemos que calcular es el menor o el mayor número que cumple con cierta propiedad.
- En estos casos para poder usar programación dinámica necesitamos asegurarnos que la solución de los subproblemas es parte de la solución de la instancia original del problema.

Principio de optimalidad

Definición

El principio de optimalidad de Bellman dice que la solución óptima de un subproblema es parte de una solución óptima del problema original.

Principio de optimalidad

Definición

El principio de optimalidad de Bellman dice que la solución óptima de un subproblema es parte de una solución óptima del problema original.

Bellman (quien descubrió la programación dinámica en 1953) afirmó que siempre que se cumple el principio de optimalidad se puede aplicar programación dinámica.

Principio de optimalidad

Definición

El principio de optimalidad de Bellman dice que la solución óptima de un subproblema es parte de una solución óptima del problema original.

Bellman (quien descubrió la programación dinámica en 1953) afirmó que siempre que se cumple el principio de optimalidad se puede aplicar programación dinámica.
Veamos algunos ejemplos de problemas en los que se cumple el principio de optimalidad y su solución con programación dinámica.

Contenidos

Ejemplos

- Monedas: Tenemos infinitas monedas con valores 1, 5, 10, 25, 50 y 100 y tenemos que pagar un valor total de n. ¿Cuál es la mínima cantidad de monedas que necesitamos?

Ejemplos

- Monedas: Tenemos infinitas monedas con valores 1, 5, 10, 25, 50 y 100 y tenemos que pagar un valor total de n. ¿Cuál es la mínima cantidad de monedas que necesitamos?
- Subrectángulos: Tenemos una matriz de $n \times m$ con valores enteros en sus casilleros, y recibimos consultas en las que se quiere averiguar la suma de los valores de un subrectángulo dado de la matriz.

Ejemplos

- Monedas: Tenemos infinitas monedas con valores 1, 5, 10, 25, 50 y 100 y tenemos que pagar un valor total de n. ¿Cuál es la mínima cantidad de monedas que necesitamos?
- Subrectángulos: Tenemos una matriz de $n \times m$ con valores enteros en sus casilleros, y recibimos consultas en las que se quiere averiguar la suma de los valores de un subrectángulo dado de la matriz.
- Algoritmo de Floyd-Warshall: Se tiene un grafo ponderado y se quieren conocer todas las distancias de cada nodo a cada uno de los demás nodos del grafo.

Monedas

- Supongamos que tenemos que calcular cuántas monedas necesitamos para pagar un valor de 1000. Si pagamos 1000 entonces al menos usamos una moneda de 1, 5, 10, 25, 50 o 100, luego quitando esta moneda sumamos un valor de 999, 995, 990, 975, 950 o 900.

Monedas

- Supongamos que tenemos que calcular cuántas monedas necesitamos para pagar un valor de 1000. Si pagamos 1000 entonces al menos usamos una moneda de 1, 5, 10, 25, 50 o 100, luego quitando esta moneda sumamos un valor de 999, 995, 990, 975, 950 o 900.
- Para todos estos valores calculamos cuántas monedas usamos como mínimo, y dentro de estos mínimos tomamos el valor mínimo y sumamos uno para obtener el resultado del problema.

Monedas

- Supongamos que tenemos que calcular cuántas monedas necesitamos para pagar un valor de 1000. Si pagamos 1000 entonces al menos usamos una moneda de 1, 5, 10, 25, 50 o 100, luego quitando esta moneda sumamos un valor de 999, 995, 990, 975, 950 o 900.
- Para todos estos valores calculamos cuántas monedas usamos como mínimo, y dentro de estos mínimos tomamos el valor mínimo y sumamos uno para obtener el resultado del problema.
- Este problema cumple el principio de optimalidad, ya que si pagamos 1000 usando una moneda de 100, quitando esta moneda vamos a pagar 900 con la mínima cantida de monedas posible.

Monedas

- La manera óptima para este caso es usar 10 monedas de 100, pero para ver que si usamos una moneda de 10 no podemos resolver el problema en menos pasos necesitamos probar qué pasa si usamos la moneda de 10.

Monedas

- La manera óptima para este caso es usar 10 monedas de 100, pero para ver que si usamos una moneda de 10 no podemos resolver el problema en menos pasos necesitamos probar qué pasa si usamos la moneda de 10.
- En este caso resolvemos el problema para un valor de 990.

Monedas

- La manera óptima para este caso es usar 10 monedas de 100, pero para ver que si usamos una moneda de 10 no podemos resolver el problema en menos pasos necesitamos probar qué pasa si usamos la moneda de 10.
- En este caso resolvemos el problema para un valor de 990.
- También tenemos que probar si usamos la moneda de 5, entonces resolvemos el problema para 995, y si volvemos a usar otra moneda de 5 en este caso volvemos a necesitar el valor del problema para 990.

Monedas

- La manera óptima para este caso es usar 10 monedas de 100, pero para ver que si usamos una moneda de 10 no podemos resolver el problema en menos pasos necesitamos probar qué pasa si usamos la moneda de 10.
- En este caso resolvemos el problema para un valor de 990.
- También tenemos que probar si usamos la moneda de 5, entonces resolvemos el problema para 995, y si volvemos a usar otra moneda de 5 en este caso volvemos a necesitar el valor del problema para 990.
- Si probamos con todas las combinaciones el algoritmo sería bastante lento, si usamos programación dinámica en cambio no necesitamos calcular más de una vez el valor de cada subproblema.

Monedas

```
1  int minimo[5000]; // Lo asumimos inicializado en -1 salvo en 0 que vale 0
2  int monedas[6]; // {1, 5, 10, 25, 50, 100}
3  int calculaMinimo(int t)
4  {
5      if(minimo[t]!=-1)
6          return minimo[t];
7      minimo[t] = t; // Sabemos que podemos usar t monedas de 1
8      for(int i=0;i<6;i++)
9      if(t-monedas[i]>=0)
10         minimo[t] = min(minimo[t],calculaMinimo(t-monedas[i]));
11     return minimo[t];
12 }
```

Subrectángulos

- Si tenemos una matriz de 1000 x 1000 y recibimos 1000 consultas, no podemos calcular todos los subrectángulos para cada consulta porque tardamos mucho. Sería bueno poder calcular rápidamente el valor de cada subrectángulo.

Subrectángulos

- Si tenemos una matriz de 1000 x 1000 y recibimos 1000 consultas, no podemos calcular todos los subrectángulos para cada consulta porque tardamos mucho. Sería bueno poder calcular rápidamente el valor de cada subrectángulo.
- Si podemos calcular el valor del subrectángulo que tiene como extremos el (0,0) y el (a, b) para cada valor de (a, b), ¿podemos calcular el valor de cualquier subrectángulo?

Subrectángulos

- Si tenemos una matriz de 1000 x 1000 y recibimos 1000 consultas, no podemos calcular todos los subrectángulos para cada consulta porque tardamos mucho. Sería bueno poder calcular rápidamente el valor de cada subrectángulo.
- Si podemos calcular el valor del subrectángulo que tiene como extremos el (0,0) y el (a, b) para cada valor de (a, b), ¿podemos calcular el valor de cualquier subrectángulo?
- Llamemosle $F(a, b)$ al valor del rectángulo con extremos en (0,0) y (a, b), entonces el valor $G(a, b, c, d)$ del rectángulo con extremos en (c, d) y (a, b) siendo $c \leq a, d \leq b$ lo podemos calcular como $F(a, b) - F(a, d - 1) - F(c - 1, b) + F(c, d)$

Subrectángulos

- Si tenemos una matriz de 1000 x 1000 y recibimos 1000 consultas, no podemos calcular todos los subrectángulos para cada consulta porque tardamos mucho. Sería bueno poder calcular rápidamente el valor de cada subrectángulo.
- Si podemos calcular el valor del subrectángulo que tiene como extremos el (0,0) y el (a, b) para cada valor de (a, b), ¿podemos calcular el valor de cualquier subrectángulo?
- Llamemosle $F(a, b)$ al valor del rectángulo con extremos en (0,0) y (a, b), entonces el valor $G(a, b, c, d)$ del rectángulo con extremos en (c, d) y (a, b) siendo $c \leq a, d \leq b$ lo podemos calcular como $F(a, b) - F(a, d - 1) - F(c - 1, b) + F(c, d)$
- En caso de que uno de los parámetros de F sea -1 simplemente devolvemos 0.

Subrectángulos

Ahora redujimos el problema a calcular para cada posición de la matriz, la suma de los valores cuyas coordenadas son menores o iguales que las de cada casillero dado.

Subrectángulos

Ahora redujimos el problema a calcular para cada posición de la matriz, la suma de los valores cuyas coordenadas son menores o iguales que las de cada casillero dado.
¿Pero cómo calculamos esto eficientemente?

Subrectángulos

Ahora redujimos el problema a calcular para cada posición de la matriz, la suma de los valores cuyas coordenadas son menores o iguales que las de cada casillero dado.
¿Pero cómo calculamos esto eficientemente?

$$G(a, b, a, b) = F(a, b) - F(a, b-1) - F(a-1, b) + F(a-1, b-1)$$

Subrectángulos

Ahora redujimos el problema a calcular para cada posición de la matriz, la suma de los valores cuyas coordenadas son menores o iguales que las de cada casillero dado.
¿Pero cómo calculamos esto eficientemente?

$$G(a, b, a, b) = F(a, b) - F(a, b-1) - F(a-1, b) + F(a-1, b-1)$$

De estos valores conocemos todos menos $F(a, b)$ ya que $G(a, b, a, b)$ es el valor de la matriz en (a, b) y como conocemos los demás valores despejamos $F(a, b)$ en esa fórmula, usando los demás valores ya calculados previamente.

Floyd-Warshall

```
1  void floyd(int n)
2  {
3      for(int t=0;t<n;t++)
4      for(int i=0;i<n;i++)
5      for(int j=0;j<n;j++)
6          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
7  }
```

```
1  void floyd(int n, int t)
2  {
3      if(t>0)
4          floyd(n,t-1);
5      for(int i=0;i<n;i++)
6      for(int j=0;j<n;j++)
7          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
8  }
9  int main()
10 {
11     ...
12     floyd(n,n-1);
13     ...
14 }
```

Floyd-Warshall

```
1  void floyd(int n)
2  {
3      for(int t=0;t<n;t++)
4      for(int i=0;i<n;i++)
5      for(int j=0;j<n;j++)
6          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
7  }
```

```
1  void floyd(int n, int t)
2  {
3      if(t>0)
4          floyd(n,t-1);
5      for(int i=0;i<n;i++)
6      for(int j=0;j<n;j++)
7          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
8  }
9  int main()
10 {
11     ...
12     floyd(n,n-1);
13     ...
14 }
```

Floyd-Warshall

```
1  void floyd(int n)
2  {
3      for(int t=0;t<n;t++)
4      for(int i=0;i<n;i++)
5      for(int j=0;j<n;j++)
6          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
7  }
```

```
1  void floyd(int n, int t)
2  {
3      if(t>0)
4          floyd(n,t-1);
5      for(int i=0;i<n;i++)
6      for(int j=0;j<n;j++)
7          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
8  }
9  int main()
10 {
11     ...
12     floyd(n,n-1);
13     ...
14 }
```

Ambos códigos son equivalentes.

Floyd-Warshall

```
1  void floyd(int n, int t)
2  {
3      if(t>0)
4          floyd(n,t-1);
5      for(int i=0;i<n;i++)
6      for(int j=0;j<n;j++)
7          d[i][j] = min(d[i][j],d
                [i][t]+d[t][j]);
8  }
9  int main()
10 {
11     ...
12     floyd(n,n-1);
13     ...
14 }
```

Floyd-Warshall

```
1  void floyd(int n, int t)
2  {
3      if(t>0)
4          floyd(n,t-1);
5      for(int i=0;i<n;i++)
6      for(int j=0;j<n;j++)
7          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
8  }
9  int main()
10 {
11     ...
12     floyd(n,n-1);
13     ...
14 }
```

Cuando llamamos a Floyd(n,t), nos queda en d[i][j] el valor del camino mínimo que va de i a j y pasa ÚNICAMENTE por los nodos $\{1, 2, \ldots, t\}$ sin contar los extremos.

Floyd-Warshall

```
1  void floyd(int n, int t)
2  {
3      if(t>0)
4          floyd(n,t-1);
5      for(int i=0;i<n;i++)
6      for(int j=0;j<n;j++)
7          d[i][j] = min(d[i][j],d
               [i][t]+d[t][j]);
8  }
9  int main()
10 {
11     ...
12     floyd(n,n-1);
13     ...
14 }
```

Cuando llamamos a Floyd(n,t), nos queda en d[i][j] el valor del camino mínimo que va de i a j y pasa ÚNICAMENTE por los nodos $\{1, 2, \ldots, t\}$ sin contar los extremos.
Esto lo podemos probar inductivamente.

Correctitud de Floyd-Warshall

- Supongamos que llamamos a Floyd(n,0). Cuando hacemos esta llamada en *d* tenemos guardadas las aristas, es decir los caminos que no usan ningún nodo en el medio.

Correctitud de Floyd-Warshall

- Supongamos que llamamos a Floyd(n,0). Cuando hacemos esta llamada en *d* tenemos guardadas las aristas, es decir los caminos que no usan ningún nodo en el medio.
- Lo primero que hacemos entonces es calcular los valores de los caminos que van de i a j pasando por 0 para todo par (i,j).

Correctitud de Floyd-Warshall

- Supongamos que llamamos a Floyd(n,0). Cuando hacemos esta llamada en *d* tenemos guardadas las aristas, es decir los caminos que no usan ningún nodo en el medio.
- Lo primero que hacemos entonces es calcular los valores de los caminos que van de i a j pasando por 0 para todo par (i,j).
- Al retornar se cumple la invariante por lo que podemos tomar a Floyd(n,0) como caso base.

Correctitud de Floyd-Warshall

- Para el paso inductivo supongamos que llamando a Floyd(n,t). Al retornar de esa llamada tenemos en d[i][j] el valor del camino mínimo de i a j, entre los que usan los nodos $\{0, \ldots, t\}$ sin contar los extremos.

Correctitud de Floyd-Warshall

- Para el paso inductivo supongamos que llamando a Floyd(n,t). Al retornar de esa llamada tenemos en d[i][j] el valor del camino mínimo de i a j, entre los que usan los nodos $\{0, \ldots, t\}$ sin contar los extremos.
- Cuando llamamos a Floyd(n,t+1) primero llamamos recursivamente a Floyd(n,t) y luego procesamos el nodo t+1.

Correctitud de Floyd-Warshall

- Para el paso inductivo supongamos que llamando a Floyd(n,t). Al retornar de esa llamada tenemos en d[i][j] el valor del camino mínimo de i a j, entre los que usan los nodos $\{0, \ldots, t\}$ sin contar los extremos.
- Cuando llamamos a Floyd(n,t+1) primero llamamos recursivamente a Floyd(n,t) y luego procesamos el nodo t+1.
- Si un camino entre a y b pasa sólo por el nodo t+1 y por nodos entre 0 y t, entonces ese camino lo consideramos, ya que si lo dividimos en la primera parte (que va del nodo a al nodo t+1) y la segunda parte (que va del t+1 al b), tenemos para ambas partes calculado el mínimo camino dadas las restricciones, luego unimos estos dos caminos y tomamos mínimo, y por lo tanto se sigue cumpliendo el invariante.

Contenidos

El juego de las piedras

Ramiro y Lucas juegan un juego. Dados dos números A y B y una bolsa con N piedras, en cada paso cada uno de ellos puede sacar 1 piedra, A piedras o B piedras. Empieza jugando Ramiro y juegan una vez cada uno. El que saca la última piedra gana. ¿Quién tiene la estrategia ganadora?

El juego de las piedras

Ramiro y Lucas juegan un juego. Dados dos números A y B y una bolsa con N piedras, en cada paso cada uno de ellos puede sacar 1 piedra, A piedras o B piedras. Empieza jugando Ramiro y juegan una vez cada uno. El que saca la última piedra gana. ¿Quién tiene la estrategia ganadora?
¿Y si el que saca la última piedra pierde?

El juego de las piedras

Ramiro y Lucas juegan un juego. Dados dos números A y B y una bolsa con N piedras, en cada paso cada uno de ellos puede sacar 1 piedra, A piedras o B piedras. Empieza jugando Ramiro y juegan una vez cada uno. El que saca la última piedra gana. ¿Quién tiene la estrategia ganadora?
¿Y si el que saca la última piedra pierde?
Este problema se puede resolver con programación dinámica. Los subproblemas para N piedras son las instancias con $N - 1$ piedras, $N - A$ piedras y $N - B$ piedras, siempre que estos números sean no negativos.

Juegos

- Muchas veces usamos programación dinámica para ver quién tiene la estrategia ganadora en un juego de dos o más jugadores.

Juegos

- Muchas veces usamos programación dinámica para ver quién tiene la estrategia ganadora en un juego de dos o más jugadores.
- En estos casos podemos usar programación dinámica porque el estado de un juego se puede reducir a cada estado según cada jugada de cada jugador.

Juegos

- Muchas veces usamos programación dinámica para ver quién tiene la estrategia ganadora en un juego de dos o más jugadores.
- En estos casos podemos usar programación dinámica porque el estado de un juego se puede reducir a cada estado según cada jugada de cada jugador.
- Esto lo podemos hacer cuando sabemos que el juego siempre termina después de una cantidad de pasos finita. Muchas veces son juegos combinatorios.

Contenidos

Juegos Combinatorios

Un juego se dice combinatorio cuando:

- Juegan dos personas

Juegos Combinatorios

Un juego se dice combinatorio cuando:

- Juegan dos personas
- Todos tienen información perfecta (saben todo lo que pasa en el juego, no hay por ejemplo cartas que no ven del otro jugador).

Juegos Combinatorios

Un juego se dice combinatorio cuando:

- Juegan dos personas
- Todos tienen información perfecta (saben todo lo que pasa en el juego, no hay por ejemplo cartas que no ven del otro jugador).
- No hay azar.

Juegos Combinatorios

Un juego se dice combinatorio cuando:

- Juegan dos personas
- Todos tienen información perfecta (saben todo lo que pasa en el juego, no hay por ejemplo cartas que no ven del otro jugador).
- No hay azar.
- El juego termina en finitos pasos con un ganador.

Juegos Combinatorios

Un juego se dice combinatorio cuando:

- Juegan dos personas
- Todos tienen información perfecta (saben todo lo que pasa en el juego, no hay por ejemplo cartas que no ven del otro jugador).
- No hay azar.
- El juego termina en finitos pasos con un ganador.
- Dada una posición y un jugador están determinadas y son siempre las mismas las posiciones a las que se puede mover el juego.