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.

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.

Problemas

Algunos problemas para practicar programación dinámica.

- http://codeforces.com/problemset/problem/225/C
- http://codeforces.com/problemset/problem/163/A

Contenidos

Subconjuntos

Subconjuntos de un conjunto

Es muy común que aparezcan problemas en los que hay que iterar sobre los subconjuntos de un conjunto, y que para calcular una función sobre un subconjunto haya que calcularla previamente sobre sus subconjuntos, para esto utilizamos programación dinámica.

Subconjuntos

Subconjuntos de un conjunto

Es muy común que aparezcan problemas en los que hay que iterar sobre los subconjuntos de un conjunto, y que para calcular una función sobre un subconjunto haya que calcularla previamente sobre sus subconjuntos, para esto utilizamos programación dinámica.

Problema de los Peces

Hay n peces en el mar. Cada un minuto se encuentran dos peces al azar (todos los pares de peces tienen la misma probabilidad) y si los peces que se encuentran son el pez i y el pez j, entonces el pez i se come al pez j con probabilidad $p[i][j]$ y el pez j se come al pez i con probabilidad $p[j][i]$. Sabemos que $p[i][j] + p[j][i] = 1$ si $i \neq j$ y $p[i][i] = 0$ para todo i. ¿Cual es la probabilidad de que sobreviva el pez 0? Sabemos que hay a lo sumo 18 peces.

Máscaras de bits

- Podemos iterar sobre los subconjuntos de un conjunto usando vectores que tengan los elementos del conjunto y crear vectores con todos los subconjuntos, pero esto es muy caro en tiempo y memoria.

Máscaras de bits

- Podemos iterar sobre los subconjuntos de un conjunto usando vectores que tengan los elementos del conjunto y crear vectores con todos los subconjuntos, pero esto es muy caro en tiempo y memoria.
- Un subconjunto de un conjunto se caracteriza por tener (1) o no tener (0) a cada elemento del conjunto.

Máscaras de bits

- Podemos iterar sobre los subconjuntos de un conjunto usando vectores que tengan los elementos del conjunto y crear vectores con todos los subconjuntos, pero esto es muy caro en tiempo y memoria.
- Un subconjunto de un conjunto se caracteriza por tener (1) o no tener (0) a cada elemento del conjunto.
- Por ejemplo, si tenemos un conjunto de 10 elementos, sus subconjuntos pueden ser representados como números entre 0 y 1023 ($2^{10} - 1$). Para cada número, si el i-ésimo bit en su representación binaria es un 1 lo interpretamos como que el i-ésimo pez del conjunto está en el subconjunto representado por ese número.

Máscaras de bits

- Cuando un número representa un subconjunto de un conjunto según los ceros o unos de su representación binaria se dice que este número es una máscara de bits

Máscaras de bits

- Cuando un número representa un subconjunto de un conjunto según los ceros o unos de su representación binaria se dice que este número es una máscara de bits
- Para ver si un número *a* representa un subconjunto del subconjunto que representa un número *b* tenemos que chequear que bit a bit si hay un 1 en *a* hay entonces un 1 en *b*

Máscaras de bits

- Cuando un número representa un subconjunto de un conjunto según los ceros o unos de su representación binaria se dice que este número es una máscara de bits
- Para ver si un número *a* representa un subconjunto del subconjunto que representa un número *b* tenemos que chequear que bit a bit si hay un 1 en *a* hay entonces un 1 en *b*
- Esto se puede chequear viendo que *a* OR *b* sea igual a *b* donde OR representa al OR bit a bit

Problema de los Peces

- Para resolver el problema de los peces, podemos tomar cada subconjunto de peces, y ver qué pasa ante cada opción de que un pez se coma a otro. Esto nos genera un nuevo subconjunto de peces para el cual resolver el problema.

Problema de los Peces

- Para resolver el problema de los peces, podemos tomar cada subconjunto de peces, y ver qué pasa ante cada opción de que un pez se coma a otro. Esto nos genera un nuevo subconjunto de peces para el cual resolver el problema.
- Cuando llegamos a un subconjunto de un sólo pez, si el pez es el pez 0, entonces la probabilidad de que sobreviva el pez 0 es 1, sino es 0.

Problema de los Peces

- Para resolver el problema de los peces, podemos tomar cada subconjunto de peces, y ver qué pasa ante cada opción de que un pez se coma a otro. Esto nos genera un nuevo subconjunto de peces para el cual resolver el problema.
- Cuando llegamos a un subconjunto de un sólo pez, si el pez es el pez 0, entonces la probabilidad de que sobreviva el pez 0 es 1, sino es 0.
- En cada paso, la probabilidad de que sobreviva el pez 0 es, la probabilidad de reducir el problema a otro subconjunto, por la probabilidad de que sobreviva dado ese subconjunto.

Problema de los Peces

```
1  double dp[1<<18];
2  int n;
3  double p[18][18];
4
5  int main()
6  {
7      forn(i,(1<<18))
8          dp[i] = -1;
9      cin >> n;
10     forn(i,n)
11     forn(j,n)
12         cin >> p[i][j];
13     printf("%6lf\n",f((1<<n)-1));
14 }
```

Problema de los Peces

```
1  double f(int mask) {
2      if(dp[mask] > -0.5) return dp[mask];
3      int vivos = 0;
4      forn(i,n) if((mask>>i)%2==1) vivos++;
5      double pares = (vivos*(vivos-1))/2;
6      if(vivos==1) {
7          if(mask==1) dp[mask] = 1.;
8          else dp[mask] = 0.;
9          return dp[mask];
10     }
11     dp[mask] = 0.;
12     forn(i,n) forn(j,i) if((mask>>i)%2==1&&(mask>>j)%2==1) {
13         if(i!=0 && j!=0) dp[mask] += (f(mask^(1<<i))*p[j][i]+f(mask^(1<<j))*
               p[i][j])/pares;
14         else if(i==0) dp[mask] += f(mask^(1<<j))*p[i][j]/pares;
15         else if(j==0) dp[mask] += f(mask^(1<<i))*p[j][i]/pares;
16     }
17     return dp[mask][pez];
18 }
```

Problema de los Peces

- Supongamos que ahora tenemos que resolver el mismo problema, pero en lugar de para un sólo pez, para todos los peces. Podríamos usar el mismo código y resolver el problema 18 veces, pero hay algo más eficiente.

Problema de los Peces

- Supongamos que ahora tenemos que resolver el mismo problema, pero en lugar de para un sólo pez, para todos los peces. Podríamos usar el mismo código y resolver el problema 18 veces, pero hay algo más eficiente.
- Lo que hicimos anteriormente fue movernos de un conjunto a sus subconjuntos. Lo que vamos a hacer ahora es movernos de un conjunto a sus superconjuntos.

Problema de los Peces

```
1  double dp[1<<18];
2  int n;
3  double p[18][18];
4
5  int main()
6  {
7      cin >> n;
8      forn(i,n)
9      forn(j,n)
10         cin >> p[i][j];
11     forn(i,(1<<n))
12         dp[i] = -1;
13     dp[(1<<n)-1] = 1.;
14     forn(i,n)
15         printf("%6lf\n",f(1<<i));
16 }
```

Problema de los Peces

```
1  double f(int mask)
2  {
3      if(dp[mask]>-0.5)
4          return dp[mask];
5      dp[mask] = 0;
6      int vivos = 1;
7      forn(i,n)
8      if((mask>>i)%2==1)
9          vivos++;
10     double pares = (vivos*(vivos-1))/2;
11     forn(i,n)
12     forn(j,n)
13     {
14         if((mask&(1<<i))!=0&&(mask&(1<<j))==0)
15             dp[mask] += f(mask|(1<<j))*p[i][j]/pares;
16     }
17     return dp[mask];
18 }
```

Contenidos

Compañeros de grupo

Enunciado

En una clase hay $2n$ alumnos ($n \leq 8$) y tienen que hacer trabajos prácticos en grupos de a 2. El i-ésimo alumno vive en el punto (x_i, y_i) de la ciudad y tarda $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ minutos en llegar a la casa del j-ésimo alumnno.

El profesor sabe que los alumnos se reunen para hacer los trabajos prácticos en la casa de uno de los dos miembros del grupo. Por eso decide que la suma de las distancias entre compañeros de grupo debe ser mínima. Dar esta distancia.

Compañeros de grupo

Enunciado

En una clase hay $2n$ alumnos ($n \leq 8$) y tienen que hacer trabajos prácticos en grupos de a 2. El i-ésimo alumno vive en el punto (x_i, y_i) de la ciudad y tarda $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ minutos en llegar a la casa del j-ésimo alumnno.
El profesor sabe que los alumnos se reunen para hacer los trabajos prácticos en la casa de uno de los dos miembros del grupo. Por eso decide que la suma de las distancias entre compañeros de grupo debe ser mínima. Dar esta distancia.

Como son 16 alumnos podemos iterar sobre los subconjuntos, en cada paso tomamos una máscara y le sacamos dos bits (que representan dos alumnos). Resolvemos el problema con la nueva máscara y le sumamos la distancia entre la casa de esos dos alumnos.

Deer Proof Fence

Enunciado

Tenemos un campo con $n \leq 16$ árboles. Sabemos las coordenadas de cada árbol y sabemos que para cada árbol necesitamos construir una cerca de modo tal que ningún ciervo se pueda acercar a menos de un metro del árbol. Si dos o más árboles están muy cerca podemos cercarlos con una misma cerca.
¿Cuántos metros de cerca necesitamos para cercar el árbol?

Deer Proof Fence

Enunciado

Tenemos un campo con $n \leq 16$ árboles. Sabemos las coordenadas de cada árbol y sabemos que para cada árbol necesitamos construir una cerca de modo tal que ningún ciervo se pueda acercar a menos de un metro del árbol. Si dos o más árboles están muy cerca podemos cercarlos con una misma cerca.
¿Cuántos metros de cerca necesitamos para cercar el árbol?

Si podemos calcular para un subconjunto de árboles cuánta cerca necesitamos para cercar a todos juntos, entonces empezamos con una máscara inicializada en unos, y probamos cada subconjunto de árboles que podemos cercar juntos, resolvemos el subproblema que resulta de sacar esos árboles, y le sumamos la cerca que usamos para ese subconjunto. De todas las formas de hacer esto nos quedamos con la que usa menos cerca.

Problemas

- http://goo.gl/iKtIH

Contenidos

El problema de la mochila - Knapsack problem

El problema de la mochila

El problema de la mochila (conocido en inglés como Knapsack problem), es un problema en el cual una persona tiene una mochila y n objetos para poner en la mochila. Cada objeto tiene un valor y un peso, y la mochila no puede cargar más que un determinado peso P. Lo que hay que calcular es el máximo valor que pueden sumar una cierta cantidad de objetos de modo tal que entren todos en simultaneo en la mochila, es decir, tal que sus pesos no excedan el valor P.

El problema de la mochila - Knapsack problem

El problema de la mochila

El problema de la mochila (conocido en inglés como Knapsack problem), es un problema en el cual una persona tiene una mochila y n objetos para poner en la mochila. Cada objeto tiene un valor y un peso, y la mochila no puede cargar más que un determinado peso P. Lo que hay que calcular es el máximo valor que pueden sumar una cierta cantidad de objetos de modo tal que entren todos en simultaneo en la mochila, es decir, tal que sus pesos no excedan el valor P.

Solución exponencial

Existe una solución en términos de n que es exponencial. Esta solución consiste en iterar sobre todos los subconjuntos de objetos, y quedarse con el de mayor valor entre aquellos que entren en la mochila. El problema es NP-completo por lo que no se conocen soluciones polinomiales hasta el momento.

Knapsack con programación dinámica

- Si bien el problema no admite soluciones polinomiales en n, existe una solución polinomial en nP, que utiliza programación dinámica y sirve siempre que P esté acotado por una constante razonable.

Knapsack con programación dinámica

- Si bien el problema no admite soluciones polinomiales en n, existe una solución polinomial en nP, que utiliza programación dinámica y sirve siempre que P esté acotado por una constante razonable.
- La solución consiste en determinar, para cada peso $0 \leq p \leq P$, el máximo valor posible que se puede obtener usando los primeros i objetos.

Knapsack con programación dinámica

- Si bien el problema no admite soluciones polinomiales en n, existe una solución polinomial en nP, que utiliza programación dinámica y sirve siempre que P esté acotado por una constante razonable.
- La solución consiste en determinar, para cada peso $0 \leq p \leq P$, el máximo valor posible que se puede obtener usando los primeros i objetos.
- Veamos una solución que asume que tenemos 100 objetos y el peso P máximo que aguanta la mochila es 1000.

Knapsack con programación dinámica

```
1  int pesos[128];
2  int valor[128];
3  int dp[128][1024];
4  int main()
5  {
6      //Leemos los pesos y valores e inicializamos dp en 0
7      for(int j=pesos[0];j;1000;j++)
8          dp[0][j] = valor[0];
9      for(int i=1;i<100;i++)
10     for(int j=pesos[i];j<1000;j++)
11         dp[i][j] = max(dp[i][j],dp[i-1][j-pesos[i]]+valor[i]);
12     cout << dp[100][1000] << endl;
13 }
```

Problemas

- http://goo.gl/ARNe7
- http://goo.gl/BJtPZ

Contenidos

Subsecuencia creciente más larga

Subsecuencia creciente más larga

El problema consiste en, dada una secuencia de números, encontrar la subsecuencia de números que aparezcan de manera creciente y que sea lo más larga posible.

Subsecuencia creciente más larga

Subsecuencia creciente más larga

El problema consiste en, dada una secuencia de números, encontrar la subsecuencia de números que aparezcan de manera creciente y que sea lo más larga posible.

Solución cuadrática

La solución trivial a este problema consiste en calcular en cada paso, la subsecuencia creciente más larga que usa los primeros i números de la secuencia, recorriendo la solución para todos los $j < i$.

Subsecuencia creciente más larga

- Existe una solución $o(n \log n)$ donde n es la longitud de la secuencia. Para implementar esa solución vamos a tener tres vectores: el input (al que llamamos seq), m y p.

Subsecuencia creciente más larga

- Existe una solución $o(n \log n)$ donde n es la longitud de la secuencia. Para implementar esa solución vamos a tener tres vectores: el input (al que llamamos seq), m y p.
- En la i-ésima iteración, vamos a tener en $m[j]$ el menor k tal que hay una subsecuencia de largo j que termina en k, con $k < i$, mientras que $m[0]$ comienza inicializado en -1.

Subsecuencia creciente más larga

- Existe una solución $o(n \log n)$ donde n es la longitud de la secuencia. Para implementar esa solución vamos a tener tres vectores: el input (al que llamamos seq), m y p.
- En la i-ésima iteración, vamos a tener en $m[j]$ el menor k tal que hay una subsecuencia de largo j que termina en k, con $k < i$, mientras que $m[0]$ comienza inicializado en -1.
- En la i-ésima posicion de p vamos a ir guardando el elemento anterior al i-ésimo en la subsecuencia creciente más larga que termina en el i-ésimo elemento del input.

Subsecuencia creciente más larga

```
1  vector<int> lis()
2  {
3      int n = seq.size(), L = 0;  m.resize(n+1);  m[0] = -1;  p.resize(n);
4      for(int i=0;i<n;i++){
5          int j = bs(L,seq[i]);
6          p[i] = m[j];
7          if(j==L||input[i]<seq[m[j+1]]){
8              m[j+1] = i;
9              L = max(L, j+1);
10         }
11     }
12     vector<int> res;
13     int t = m[L];
14     while(t!=-1){
15         res.push_back(seq[t]);
16         t = p[t];
17     }
18     reverse(res.begin(),res.end());
19     return res;
20 }
```

Subsecuencia creciente más larga

```
1  int bs(int L, int num)
2  {
3      int mx = L+1, mn = 0;
4      while(mx-mn>1)
5      {
6          int mid = (mx+mn)/2;
7          if(seq[m[mid]]<num)
8              mn = mid;
9          else
10             mx = mid;
11     }
12     return mn;
13 }
```

Subsecuencia creciente más larga

- Observemos que m va a estar definido para todo $j < L$ y que $m[j]$ está en el rango de vec para todo $0 < j \leq L$. Además en la búsqueda binaria nunca evaluamos $seq[m[0]]$.

Subsecuencia creciente más larga

- Observemos que m va a estar definido para todo $j < L$ y que $m[j]$ está en el rango de vec para todo $0 < j \leq L$. Además en la búsqueda binaria nunca evaluamos $seq[m[0]]$.
- También podemos ver que p va a estar siempre definido para todo $0 \leq i < n$.

- http://www.spoj.com/problems/SUPPER/
- http://www.spoj.com/problems/LIS2/

Contenidos

Minimum Vertex Cover

Minimum Vertex Cover

Es muy común que haya problemas que para grafos generales son difíciles de resolver, pero que pueden ser resueltos polinomialmente sobre árboles. Este es el caso de Minimum Vertex Cover. Este problema consiste en encontrar un conjunto de vértices lo más chico posible de modo tal de que todo vértice, o bien está en el conjunto, o bien es adyacente a un vértice del conjunto.

Minimum Vertex Cover

Idea de la solución:

- Las hojas del árbol nunca son parte del Minimum Vertex Cover.

Minimum Vertex Cover

Idea de la solución:

- Las hojas del árbol nunca son parte del Minimum Vertex Cover.
- Dado un nodo que no es hoja, si ninguno de sus hijos está en el Minimum Vertex Cover, entonces el nodo sí está, caso contrario no está.

Minimum Vertex Cover

```
1  vector<vector<int> > hijos;
2  vector<int> dp,dp2;
3
4  int calc(int t)
5  {
6      dp2[t] = 1;
7      dp[t] = 0;
8      for(int i=0;i<hijos[t].size();i++)
9      {
10         dp[t] += calc(hijos[t][i]);
11         dp2[t] = max(dp2[t],dp2[hijos[t][i]]+1);
12     }
13     if(dp2[t]>1)
14     {
15         dp2[t] = 0;
16         dp[t]++;
17     }
18     return dp[t];
19 }
```

Contenidos

Maximum Independent Subset

Maximum Independent Subset

El problema Maximum Independent Subset (también conocido como máximo subconjunto independiente) consiste en, dado un grafo, encontrar el conjunto más grande de vértices tales que no haya dos vértices en el subconjunto que tengan un eje entre sí.

Maximum Independent Subset

Maximum Independent Subset

El problema Maximum Independent Subset (también conocido como máximo subconjunto independiente) consiste en, dado un grafo, encontrar el conjunto más grande de vértices tales que no haya dos vértices en el subconjunto que tengan un eje entre sí.

Este problema tiene una solución polinomial en árboles.

Maximum Independent Subset

- Para resolver el problema vamos a tener dos estados. $dp1[t]$ indica el mayor conjunto independiente del subárbol con raiz t y que contiene a t. $dp2[t]$ indica el mayor conjutno independiente del subárbol con raiz t y que no contiene a t.

Maximum Independent Subset

- Para resolver el problema vamos a tener dos estados. $dp1[t]$ indica el mayor conjunto independiente del subárbol con raiz t y que contiene a t. $dp2[t]$ indica el mayor conjutno independiente del subárbol con raiz t y que no contiene a t.
- La solución al problema es el máximo entre $dp1$ y $dp2$ evaluados en la raiz del árbol.

Maximum Independent Subset

```
1  vector<vector<int> > hijos;
2  vector<int> dp1, dp2;
3  int calc1(int t);
4  int calc2(int t){
5      dp2[t] = 0;
6      for(int i=0;i<hijos[t].size();i++){
7          calc1(hijos[t][i]);
8          dp2[t] += dp1[hijos[t][i]];
9      }
10     return dp2[t];
11 }
12 int calc1(int t){
13     dp1[t] = 1;
14     for(int i=0;i<hijos[t].size();i++){
15         calc2(hijos[t][i]);
16         dp1[t] += dp2[hijos[t][i]];
17     }
18     return dp1[t];
19 }
```