Programación Dinámica

Quimey Vivas
~~Agustín Santiago Gutiérrez~~

Resuelvo problemas por comida
~~Facultad de Ciencias Exactas y Naturales~~
~~Universidad de Buenos Aires~~

Training camp Neuquén ACM-ICPC 2016

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 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 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
- Básicamente tenemos “problemas de memoria”.

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
- Básicamente tenemos “problemas de memoria”.
- ¿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
- Básicamente tenemos “problemas de memoria”.
- ¿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

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 dos o más subproblemas o reducirlo a una instancia más fácil de calcular del problema.

Programación dinámica

Visión tradicional:

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

Visión tradicional:

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

Visión tradicional:

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: 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.
- 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)

- Este algoritmo no utiliza memoización (según algunos autores, esta forma *bottom-up* es programación dinámica, pero memoización no).
- Esta forma es mucho más eficiente que memoización, cuando se calculan la mayoría de las entradas de la tabla cache.

Superposición de subproblemas

- Los dos ejemplos vistos presentan una característica típica de la programación dinámica: La 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

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.
- 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.

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.
- 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.

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.
- 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.

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.
- 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.

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.
- 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.

Resumiendo

- Hemos visto dos maneras de implementarl programación dinámica:
  - 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

Problemas

Algunos problemas para practicar programación dinámica.

- http://codeforces.com/problemset/problem/225/C
- http://codeforces.com/problemset/problem/163/A
- http://goo.gl/ARNe7
- http://goo.gl/BJtPZ

Contenidos

Introducción

- El patrón de “Dinámicas en rangos” es un patrón típico que aparece mucho.
- Consiste en estados (o subproblemas) de la forma (a, b), es decir, 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.

Parentesis

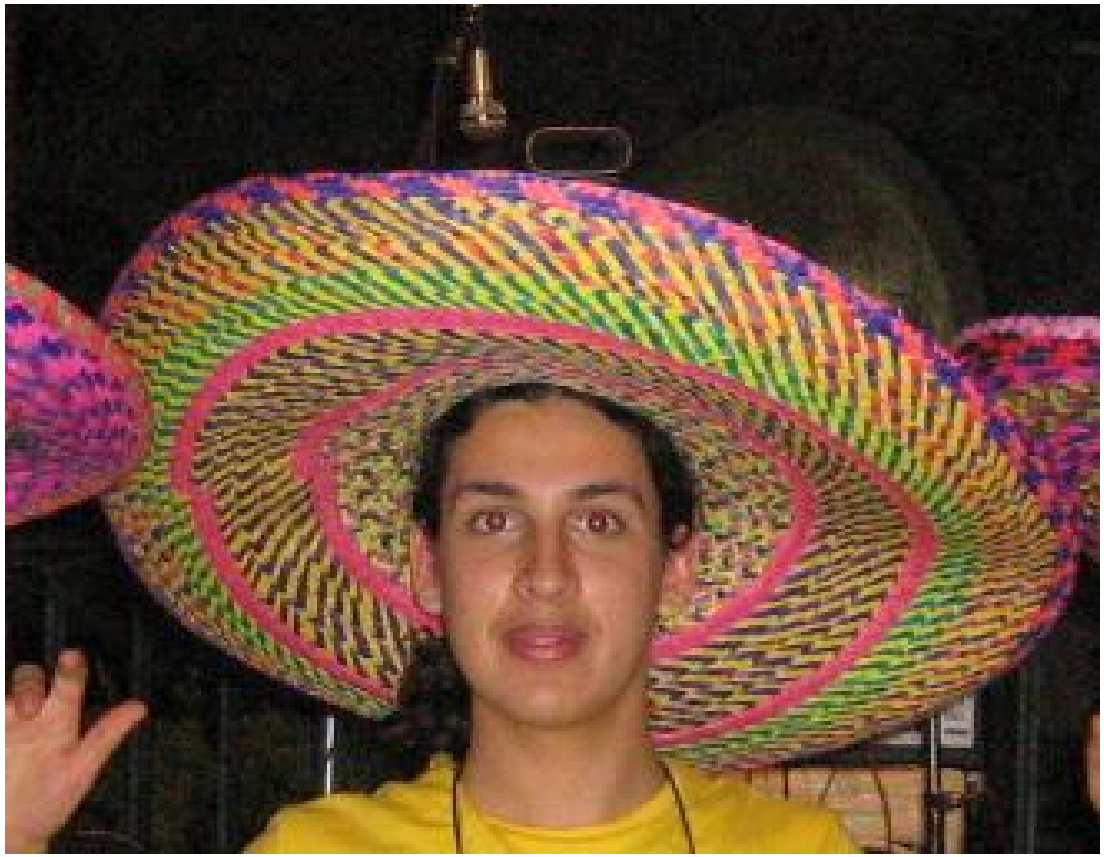
- Dada una cadena de caracteres {, }, [,] , (y), de longitud par, dar la mínima cantidad de reemplazos de caracteres que se le deben realizar a este string para dejar una secuencia “bien parenteseada”.
- Una secuencia bien parenteseada T se puede formar con las siguientes reglas a partir de secuencias bien parenteseadas S y S_2:
  - $T = \emptyset$
  - $T = SS_2$
  - $T = (S)$
  - $T = [S]$
  - $T = \{S\}$

Contenidos

Tarea

Problema de la IOI de México 2006

- `http://ioinformatics.org/locations/ioi06/contest/day2/mexico/mexico.pdf`



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. ¿Cuál 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.~~ ¿Cuán caro es?

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.~~ ¿Cuán caro es?
- 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.~~ ¿Cuán caro es?
- 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];
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.
- Ejercicio: Pensar algo mejor para evitar tener que repetir el cálculo de la tabla de DP para cada pez.

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.
- Ejercicio: Pensar algo mejor para evitar tener que repetir el cálculo de la tabla de DP para cada pez.

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 alumno.

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 alumno.
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.

Problema del viajante de comercio

Enunciado

Un viajante debe recorrer $n \leq 20$ ciudades exactamente una vez cada una, formando un ciclo para vender sus productos en cada ciudad. Conociendo la tabla de distancias entre cada par de ciudades, proponer un ciclo que minimice la longitud total recorrida.

Problema del viajante de comercio

Enunciado

Un viajante debe recorrer $n \leq 20$ ciudades exactamente una vez cada una, formando un ciclo para vender sus productos en cada ciudad. Conociendo la tabla de distancias entre cada par de ciudades, proponer un ciclo que minimice la longitud total recorrida.

Podemos encontrar una solución $O(2^n n^2)$ utilizando programación dinámica. Notar que esto es **muchísimo** mejor que el sencillo $O(n!)$. Como estado podemos tomar la ciudad actual, y el conjunto (máscara) de las ciudades ya recorridas, asumiendo que se empezó de una ciudad fija arbitraria.

n	$2^n n^2$	$n!$
5	800	120
10	102400	3628800
15	7372800	1307674368000
20	419430400	2432902008176640000

Cubrir un tablero con dominos

Enunciado

Dado un tablero de $2 \times n$, calcular la cantidad de formas de cubrilo con fichas de domino (1×2 o 2×1). ¿Qué pasa si el tablero es de $3 \times n$? ¿Y $k \times n$?

Tarea

- Tutorial de topcoder: *A bit of fun: fun with bits*, por Bruce Merry
  `http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=bitManipulation`
- `https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2451`
- `http://goo.gl/iKtIH`