Programación Dinámica + Backtracking

Problemas, Algoritmos y Programación

Agosto de 2016

Contenidos

Contenidos

Disclaimer importante!

Estas diapositivas no son autocontenidas: contienen mucho pseudocódigo no explicado, el cual será explicado en pizarrón durante la clase.

Programación Dinámica

Programación dinámica... ¿Y eso con qué se come?

Programación Dinámica

Programación dinámica... ¿Y eso con qué se come?

Repasemos un poco lo que vimos en Algoritmos 3!

Programación Dinámica

Programación Dinámica

Programación dinámica es una técnica algorítmica que utilizamos cuando queremos resolver un problema recursivo sin repetir los mismos cálculos más de una vez.

Programación Dinámica

Programación Dinámica

Programación dinámica es una técnica algorítmica que utilizamos cuando queremos resolver un problema recursivo sin repetir los mismos cálculos más de una vez.

Ejemplos: Calcular los números de Fibonacci, los números combinatorios, subsecuencia común más larga (LCS), subsecuencia creciente más larga (LIS).

Programación Dinámica

Programación Dinámica

Programación dinámica es una técnica algorítmica que utilizamos cuando queremos resolver un problema recursivo sin repetir los mismos cálculos más de una vez.

Ejemplos: Calcular los números de Fibonacci, los números combinatorios, subsecuencia común más larga (LCS), subsecuencia creciente más larga (LIS).
Los primeros dos ya los deben haber visto el Algo 3 y son bastante sencillos, veamos un poco los otros dos para repasar.

Longest Common Subsequence

Definición del problema

El problema de subsecuencia común más larga (o Longest Common Subsequence) consiste en, dadas dos secuencias (de números, de caracteres, de strings, etc), encontrar la secuencia más larga que sea subsecuencia de ambas.

Longest Common Subsequence

Definición del problema

El problema de subsecuencia común más larga (o Longest Common Subsequence) consiste en, dadas dos secuencias (de números, de caracteres, de strings, etc), encontrar la secuencia más larga que sea subsecuencia de ambas.

Existen muchas subsecuencias y no podemos probar todas. Veamos la idea en el pizarrón (definiendo bien qué es una subsecuencia) y en la próxima diapositiva un pseudocódigo de la solución.

Longest Common Subsequence

```
int LCS (int[] s1, int[] s2):
    int[size(s1)+1][size(s2)+1] sub
    for i: 0 -> size(s1)+1
        sub[i][0] = 0
    for j: 0 -> size(s2)+1
        sub[0][j] = 0
    for i: 1 -> size(s1)+1
    for j: 1 -> size(s2)+1
        sub[i][j] = max(sub[i-1][j],sub[i][j-1])
        if s1[i-1] == s2[j-1]
            sub[i][j] = max(sub[i][j],sub[i-1][j-1]+1)
    return sub[size(s1)][size(s2)]
```

Longest Common Subsequence

```
int LCS (int[] s1, int[] s2):
    int[size(s1)+1][size(s2)+1] sub
    for i: 0 -> size(s1)+1
        sub[i][0] = 0
    for j: 0 -> size(s2)+1
        sub[0][j] = 0
    for i: 1 -> size(s1)+1
    for j: 1 -> size(s2)+1
        sub[i][j] = max(sub[i-1][j],sub[i][j-1])
        if s1[i-1] == s2[j-1]
            sub[i][j] = max(sub[i][j],sub[i-1][j-1]+1)
    return sub[size(s1)][size(s2)]
```

Con este algoritmo encontramos la longitud de la subsecuencia. Para reconstruirla podemos guardar más informacion en sub que sólo la longitud (queda como ejercicio!).

Longest Increasing Subsequence

Definición del problema

El problema de subsecuencia creciente más larga (o Longest Increasing Subsequence) consiste en, dada una secuencia (de números, caracteres, o cualquier cosa que tenga definido un orden), encontrar la subsecuencia más larga que sea creciente.

Longest Increasing Subsequence

Definición del problema

El problema de subsecuencia creciente más larga (o Longest Increasing Subsequence) consiste en, dada una secuencia (de números, caracteres, o cualquier cosa que tenga definido un orden), encontrar la subsecuencia más larga que sea creciente.

Una idea clásica en programación dinámica, cuando hay que resolver problemas sobre un arreglo, es ir resolviendo el problema sobre subarreglos (y en muchos casos, como en este problema, sobre prefijos). Veamos la idea en el pizarrón, y luego el pseudocódigo

Longest Common Subsequence

```
int[] LIS (int[] s):
    int[size(s)] p = [-1 * size(s)]
    int[size(s)+1] m = [-1 * size(s)]
    m[1] = 0
    int L = 1
    for i: 1 -> size(s)
        for j: 1 -> L+1
            if s[i] > s[m[j]]
                p[i] = m[j]
                if L == j or s[i] < s[m[j+1]]
                    m[j+1] = i
                    L = max(L,j+1)
```

Longest Common Subsequence

```
int[] LIS (int[] s):
    int[size(s)] p = [-1 * size(s)]
    int[size(s)+1] m = [-1 * size(s)]
    m[1] = 0
    int L = 1
    for i: 1 -> size(s)
        for j: 1 -> L+1
            if s[i] > s[m[j]]
                p[i] = m[j]
                if L == j or s[i] < s[m[j+1]]
                    m[j+1] = i
                    L = max(L,j+1)
```

Con esto calculamos p y m, en la próxima diapositiva seguimos (porque no entraba todo en una!)

Longest Common Subsequence

```
int[L] res
res[L-1] = m[L]
L--
while L > 0
    res[L-1] = p[res[L]]
    L--
return res
```

Longest Common Subsequence

```
int[L] res
res[L-1] = m[L]
L--
while L > 0
    res[L-1] = p[res[L]]
    L--
return res
```

Si bien el pseudocódigo está en las diapositivas, la explicación del algoritmo la hacemos en pizarrón.

Longest Common Subsequence

```
int[L] res
res[L-1] = m[L]
L--
while L > 0
    res[L-1] = p[res[L]]
    L--
return res
```

Si bien el pseudocódigo está en las diapositivas, la explicación del algoritmo la hacemos en pizarrón.
Este algoritmo tiene complejidad $\mathcal{O}(N^2)$.

Longest Common Subsequence

```
int[L] res
res[L-1] = m[L]
L--
while L > 0
    res[L-1] = p[res[L]]
    L--
return res
```

Si bien el pseudocódigo está en las diapositivas, la explicación del algoritmo la hacemos en pizarrón.
Este algoritmo tiene complejidad $\mathcal{O}(N^2)$.
Se puede bajar a $\mathcal{O}(N \log N)$ aprovechando que $s[m[j]] > s[m[j-1]]$, y al ser creciente podemos reemplazar una búsqueda lineal por una búsqueda binaria.

Multiplicación de matrices

Un último ejemplo de repaso y vamos a lo nuevo: Cómo multiplicar matrices eficientemente.

Multiplicación de matrices

Un último ejemplo de repaso y vamos a lo nuevo: Cómo multiplicar matrices eficientemente.

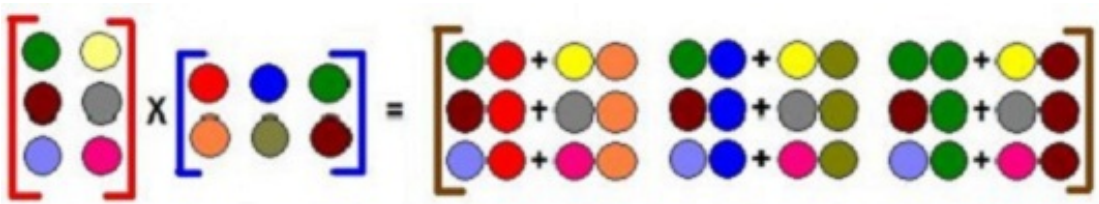
Una matriz es un arreglo bidimensional con numeritos. ¿Se acuerdan cómo multiplicar matrices?

Cómo multiplicar matrices

Veamos un ejemplo con colores (los ejemplos con colores siempre ayudan!)

Cómo multiplicar matrices

Veamos un ejemplo con colores (los ejemplos con colores siempre ayudan!)



Cómo multiplicar dos matrices

Multipliquemos $M_1 \in \mathbb{R}^{a \times b}$ por $M_2 \in \mathbb{R}^{b \times c}$

```
int[][] mult(int[][] M1, int[][] M2, int a, int b, int c):
    int[a][c] res
    for i: 0 -> a
        for j: 0 -> c
            res[i][j] = 0
            for t: 0 -> b
                res[i][j] += M1[i][t] * M2[t][j]
    return res
```

Cómo multiplicar dos matrices

Multipliquemos $M_1 \in \mathbb{R}^{a \times b}$ por $M_2 \in \mathbb{R}^{b \times c}$

```
int[][] mult(int[][] M1, int[][] M2, int a, int b, int c):
    int[a][c] res
    for i: 0 -> a
        for j: 0 -> c
            res[i][j] = 0
            for t: 0 -> b
                res[i][j] += M1[i][t] * M2[t][j]
    return res
```

El costo de esta operación es $\mathcal{O}(abc)$.

¿Y si tenemos muchas matrices?

Veamos qué pasa cuando en vez de dos matrices tenemos tres:

$$M_1(\in \mathbb{R}^{a \times b}) \times M_2(\in \mathbb{R}^{b \times c}) \times M_3(\in \mathbb{R}^{c \times d})$$

¿Y si tenemos muchas matrices?

Veamos qué pasa cuando en vez de dos matrices tenemos tres:

$$M_1(\in \mathbb{R}^{a\times b}) \times M_2(\in \mathbb{R}^{b\times c}) \times M_3(\in \mathbb{R}^{c\times d})$$

El costo puede ser $abc + acd$ si multiplicamos $(M_1 M_2)M_3$ o $bcd + abd$ si multiplicamos $M_1(M_2 M_3)$

¿Y si tenemos muchas matrices?

Veamos qué pasa cuando en vez de dos matrices tenemos tres:

$$M_1(\in \mathbb{R}^{a\times b}) \times M_2(\in \mathbb{R}^{b\times c}) \times M_3(\in \mathbb{R}^{c\times d})$$

El costo puede ser $abc + acd$ si multiplicamos $(M_1 M_2)M_3$ o $bcd + abd$ si multiplicamos $M_1(M_2 M_3)$
Con 3 matrices tenemos 2 formas de hacer los productos, con N matrices hay $(N-1)!$ formas distintas. Decidir cuál es la más rápida de todas es un problema de programación dinámica.

Contenidos

¿Qué es una máscara bits?

Una máscara de bits es un vector de ceros y unos, almacenados en los bits de un número entero, que se usa generalmente para representar conjuntos (0 si no está, 1 si está).

¿Qué es una máscara bits?

Una máscara de bits es un vector de ceros y unos, almacenados en los bits de un número entero, que se usa generalmente para representar conjuntos (0 si no está, 1 si está).
Cuando representamos conjuntos podemos hacer muchas operaciones de manera eficiente:

$$\begin{aligned} A \cap B &= m_A \,\&\, m_B \\ A \cup B &= m_A | m_B \\ A \setminus B &= m_A \,\&\, \sim m_B \\ A \triangle B &= m_B \,\hat{}\, m_B \\ A \subseteq B &\Leftrightarrow m_A \,\&\, m_B = m_A \end{aligned}$$

($\triangle$ es la diferencia simétrica, los que están en A y no en B o están en B y no en A)

Programación dinámica con máscaras de bits

Veamos un ejemplo:

Grupos de TPs

En la materia Algoritmos y Estructuras de Datos VIII[1], los TPs son de a 2. Afortunadamente, la cantidad de alumnos N de la materia es par. Los alumnos se juntarán a hacer TP en la casa de uno de los dos integrantes, y queremos minimizar la suma del tiempo que tarden viajando. Recibimos una matriz simétrica $M \in \mathbb{N}^{N \times N}$ que en $M_{i,j}$ tiene el tiempo que tarda en llegar el alumno i a la casa del alumno j, y nos piden calcular esta suma mínima.

[1] Materia obligatoria para la Licenciatura en Ciencias de la Computación según el plan vigente a partir de 2018.

Grupos de TPs

```
int respuesta[1<<N] //inicializado en -1
int tiempos2(int[][] M, int N, int mask):
  if respuesta[mask] != -1
    return respuesta[mask]
  int res = INF
  for i: 0->N
  for j: 0-> i
  if (mask & ((1<<i) | (1<<j))) ==  ((1<<i) | (1<<j))
    res = min(res,M[i][j]+tiempos2(M,N,mask^((1<<i)|(1<<j))))
  respuesta[mask] = res
  return res
int tiempos(int[][] M, int N):
  respuesta[0] = 0
  return tiempos2(M,N,(1<<N)-1)
```

$1 << N$ es el número 2^N, INF es un número suficientemente grande para que siempre sea cota superior de la respuesta.

Iterando sobre los subconjuntos

A veces no alcanza sólo con ver algunos subconjuntos (en el ejemplo anterior los de tamaño 2), sino que queremos iterar sobre todos los subconjuntos de un conjunto. Veamos un problema que sirva como ejemplo:

Iterando sobre los subconjuntos

A veces no alcanza sólo con ver algunos subconjuntos (en el ejemplo anterior los de tamaño 2), sino que queremos iterar sobre todos los subconjuntos de un conjunto. Veamos un problema que sirva como ejemplo:

La estrella de la muerte

Darth Vader terminó de construir su nueva estrella de la muerte. Al finalizar esta construcción descubrió que un disparo de su estrella apuntando a la posición (x, y) destruye a todos los planetas que se encuentren en el cuadrado $[x - T, x + T] \times [y - T, y + T]$. ¿Cuántos disparos necesita el poderoso Vader para destruir los N planetas de sus enemigos?

Darth Vader

Iterando sobre los subconjuntos

```
bool entranJuntos[1<<N]
int disparos[1<<N] //inicializado en -1
int minDisparos2(int N, int mask):
    if disparos[mask] != -1
        return disparos[mask]
    int res = N
    for i: 1 -> (1<<N)
        if (i & mask) == i and entranJuntos[i]
            res = min(res,minDisparos2(N,mask^i)+1)
    disparos[mask] = res
    return res
int minDisparos(int N):
    disparos[0] = 0
    return minDisparos2(N,(1<<N)-1)
```

Asumimos que ya calculamos en *entranJuntos*[*mask*] si los planetas representados por la máscara mask entran todos juntos en un mismo disparo.

Complejidad

Tenemos 2^N instancias, cada una de las cuales tiene un for que itera sobre 2^N valores, luego la complejdad es $\mathcal{O}(4^N)$

Complejidad

Tenemos 2^N instancias, cada una de las cuales tiene un for que itera sobre 2^N valores, luego la complejdad es $\mathcal{O}(4^N)$

Esta complejidad puede ser mejorada si en vez de iterar sobre todos los conjuntos, en la instancia en la que calculamos los subconjuntos de *mask* iteramos únicamente sobre los subconjuntos de *mask*.

Complejidad

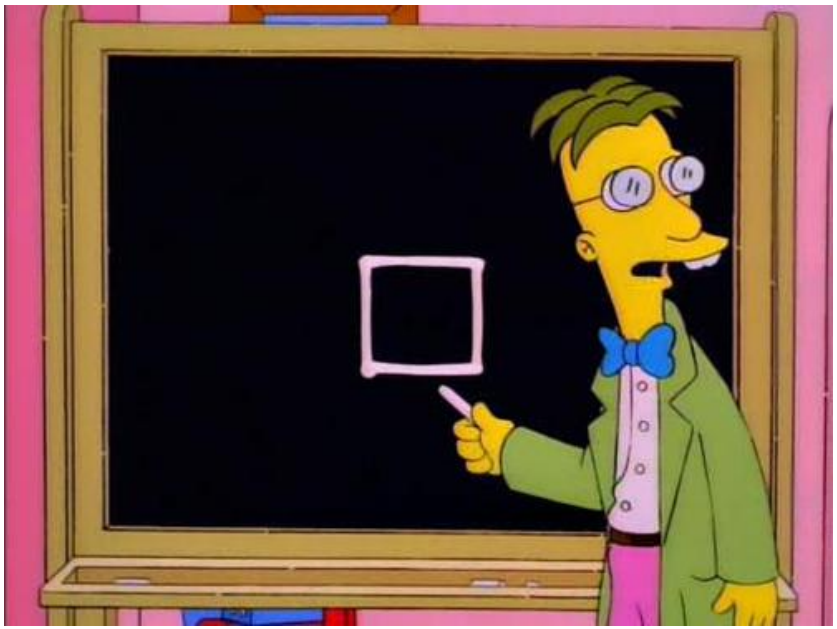
Tenemos 2^N instancias, cada una de las cuales tiene un for que itera sobre 2^N valores, luego la complejidad es $\mathcal{O}(4^N)$

Esta complejidad puede ser mejorada si en vez de iterar sobre todos los conjuntos, en la instancia en la que calculamos los subconjuntos de *mask* iteramos únicamente sobre los subconjuntos de *mask*.

Hay $\binom{N}{i}$ subconjuntos de i elementos, luego la complejidad queda

$$\mathcal{O}\left(\sum_{i=0}^{N}\binom{N}{i}2^i\right) = \mathcal{O}(3^N)$$

Más despacio cerebrito



¿Y de dónde sale ese 3^N mágico?

Más despacio cerebrito

Hay (al menos) dos formas de ver esto:

Más despacio cerebrito

Hay (al menos) dos formas de ver esto:

- $3^N = (2+1)^N = \sum_{i=0}^{N} \binom{N}{i} 2^i 1^{N-i}$ (Nuestro viejo y querido binomio de Newton!).

Más despacio cerebrito

Hay (al menos) dos formas de ver esto:

- $3^N = (2+1)^N = \sum_{i=0}^{N} \binom{N}{i} 2^i 1^{N-i}$ (Nuestro viejo y querido binomio de Newton!).
- Tenemos 2^N conjuntos, estamos procesando el conjunto C_{mask} y queremos iterar sobre $C_i \subseteq C_{mask} \subseteq C_{2^N-1}$ (Llamemosle $C_{2^N-1} = C$), cada iteración del for en alguna de las llamadas a la función que calcula el resultado se mapea con esta terna (C_i, C_{mask}, C), luego la complejidad es $\mathcal{O}(T)$ donde T es la cantidad de formas de elegir i y $mask$. Cada bit tiene tres opciones:
  - Está prendido en $2^N - 1$ y apagado en $mask$ y en i.
  - Está prendido en $2^N - 1$ y $mask$ pero apagado en i.
  - Está prendido en $2^N - 1$, en $mask$ y en i.

  Luego hay 3 opciones para cada uno de los N bits: 3^N

¿Y ahora cómo hacemos?

La teoría es muy linda pero... ¿cómo iteramos sólamente sobre los subconjuntos de *mask*?

¿Y ahora cómo hacemos?

La teoría es muy linda pero... ¿cómo iteramos sólamente sobre los subconjuntos de *mask*?

En la próxima diapositiva: magia en estado puro!

¿Y ahora cómo hacemos?

```
for(int i=mask; i != 0; i = mask&(i-1))
```

¿Y ahora cómo hacemos?

```
for(int i=mask; i != 0; i = mask&(i-1))
```

¿Y esto qué hace?

¿Y ahora cómo hacemos?

```
for(int i=mask; i != 0; i = mask&(i-1))
```

¿Y esto qué hace?

Este for empieza en *mask*, y va iterando, de mayor a menor, en las máscaras que representan los subconjuntos de *mask*, hasta llegar al subconjunto vacío (el cual no lo itera).

¿Y ahora cómo hacemos?

```
for(int i=mask; i != 0; i = mask&(i-1))
```

¿Y esto qué hace?

Este for empieza en *mask*, y va iterando, de mayor a menor, en las máscaras que representan los subconjuntos de *mask*, hasta llegar al subconjunto vacío (el cual no lo itera).

Al hacer and con *mask* es fácil ver que i siempre representa un subconjunto de *mask*. Al hacer and con $i - 1$ es fácil ver que i pasa a ser un subconjunto de $i - 1$, y luego siempre decrece, sólo nos falta ver que los recorre todos (es decir, que va decreciendo sin saltearse ninguno).

¿Y ahora cómo hacemos?

```
for(int i=mask; i != 0; i = mask&(i-1))
```

i empieza en un subconjunto de $mask$. $i - 1$ representa otro conjunto, que tiene todos los bits de i prendidos, a excepción del menos significativo.

¿Y ahora cómo hacemos?

```
for(int i=mask; i != 0; i = mask&(i-1))
```

i empieza en un subconjunto de *mask*. $i - 1$ representa otro conjunto, que tiene todos los bits de i prendidos, a excepción del menos significativo.

Al cambiar i por $i - 1$ sacamos de nuestro conjunto al elemento "más chico"de *mask* que hay en i y prendemos todos los que son "menores" a ese, luego con and *mask* apagamos los que no pertenecen a *mask*, y nos queda el siguiente conjunto más chico de *mask*.

¿Y ahora cómo hacemos?

```
for(int i=mask; i != 0; i = mask&(i-1))
```

i empieza en un subconjunto de *mask*. $i - 1$ representa otro conjunto, que tiene todos los bits de i prendidos, a excepción del menos significativo.

Al cambiar i por $i - 1$ sacamos de nuestro conjunto al elemento "más chico"de *mask* que hay en i y prendemos todos los que son "menores" a ese, luego con and *mask* apagamos los que no pertenecen a *mask*, y nos queda el siguiente conjunto más chico de *mask*.

Veamos bien esto con un ejemplo en el pizarrón.

Contenidos

Backtracking

¿Qué era backtracking?

Backtracking es una técnica algorítmica que va construyendo soluciones paso por paso, pero a diferencia de los algoritmos golosos o de programación dinámica, no reutiliza información ni calcula más de una vez las mismas instancias.

Backtracking

¿Qué era backtracking?

Backtracking es una técnica algorítmica que va construyendo soluciones paso por paso, pero a diferencia de los algoritmos golosos o de programación dinámica, no reutiliza información ni calcula más de una vez las mismas instancias.

En criollo: Prueba todos con todos.

¿Y eso para qué se usa?

Al probar "todos con todos" la complejidad del backtracking, en peor caso, suele ser exponencial, por lo que suele ser utilizado en problemas para los cuales no se conocen soluciones polinomiales (por ejemplo, problemas NP-completos).

¿Y eso para qué se usa?

Al probar "todos con todos" la complejidad del backtracking, en peor caso, suele ser exponencial, por lo que suele ser utilizado en problemas para los cuales no se conocen soluciones polinomiales (por ejemplo, problemas NP-completos).

En Algoritmos III hicieron foco en las podas, que son lo que distinguen al backtracking de la fuerza bruta común y corriente, y que hacen que baje la constante del algoritmo, y para algunos casos también la complejidad.

¿Y eso para qué se usa?

Al probar "todos con todos" la complejidad del backtracking, en peor caso, suele ser exponencial, por lo que suele ser utilizado en problemas para los cuales no se conocen soluciones polinomiales (por ejemplo, problemas NP-completos).

En Algoritmos III hicieron foco en las podas, que son lo que distinguen al backtracking de la fuerza bruta común y corriente, y que hacen que baje la constante del algoritmo, y para algunos casos también la complejidad.

En esta materia no vamos a andar concentrarnos en las podas (sí! pueden no usar podas si hacen un backtracking en el TP!) que hacen un poquito más rápido el algoritmo: Vamos a ver una técnica que hace MUCHO más rápido el algoritmo.

Pero... ¿cómo era eso del backtracking?

Nuevamente, hagamos un repaso a través un ejemplo:

Coloreo

Se quieren pintar los vértices de un grafo de modo tal que todo par de vecinos sean de distinto color. ¿Cuántos colores se necesitan como mínimo?

Pero... ¿cómo era eso del backtracking?

Nuevamente, hagamos un repaso a través un ejemplo:

Coloreo

Se quieren pintar los vértices de un grafo de modo tal que todo par de vecinos sean de distinto color. ¿Cuántos colores se necesitan como mínimo?

Veamos como resolver el ejemplo a través del pseudocódigo que nos dice si podemos pintar G con C colores.

Coloreo

```
bool pintar(Grafo G, Mapeo M, int C, int v):
    if v == |V(G)|
        return true
    for i: 0 -> C:
        if(ningunVecinoDeColor(G,M,v,i))
            M[v] = i
            if(pintar(G,M,C,v+1))
                return true
            M.borrar(v)
    return false
bool coloreo(Grafo G, int C):
    Mapeo M(int -> int) = vacio
    return pintar(G,M,C,0)
```

ningunVecinoDeColor(G, M, v, i) devuelve true sii v no tiene vecinos en G de color i en el mapeo M.

¿Queremos toda la información?

Cuando la respuesta es binaria (como en coloreo) y no hay una forma de ordenar las posibles respuestas, no nos queda otra que hacer backtracking común y corriente, pero cuando la respuesta es un número, y queremos un máximo, un mínimo, o alguna otra función que dependa de algún orden, podemos hacer algo mejor.

¿Queremos toda la información?

Cuando la respuesta es binaria (como en coloreo) y no hay una forma de ordenar las posibles respuestas, no nos queda otra que hacer backtracking común y corriente, pero cuando la respuesta es un número, y queremos un máximo, un mínimo, o alguna otra función que dependa de algún orden, podemos hacer algo mejor.

Esta nueva técnica que vamos a utilizar se basa en calcular TODAS las posibles respuestas para instancias más chicas (que las necesitamos para calcular la respuesta de una instancia más grande), pero para la instancia original del problema sólo buscamos si podemos obtener UNA: la que nos interesa.

¿Queremos toda la información?

Cuando la respuesta es binaria (como en coloreo) y no hay una forma de ordenar las posibles respuestas, no nos queda otra que hacer backtracking común y corriente, pero cuando la respuesta es un número, y queremos un máximo, un mínimo, o alguna otra función que dependa de algún orden, podemos hacer algo mejor.

Esta nueva técnica que vamos a utilizar se basa en calcular TODAS las posibles respuestas para instancias más chicas (que las necesitamos para calcular la respuesta de una instancia más grande), pero para la instancia original del problema sólo buscamos si podemos obtener UNA: la que nos interesa.

Esta técnica se conoce como Meet in the Middle, y su nombre viene de que resolvemos dos mitades por separado, y después vemos como las pegamos en el medio.

Contenidos

Meet in the Middle

Veamos como venimos haciendo hasta ahora, la definición a través de un ejemplo.

Paridad

Tenemos N números A_i enteros entre 0 y $2^M - 1$ inclusive. Definimos la función paridad(X) como la cantidad de bits en 1 de X módulo 2. Por ejemplo: 0, 3, 5, 6 y 9 tienen paridad 0 (ya que tienen 0 o 2 bits en 1), mientras que 1, 2, 4, 7 y 8 tienen paridad 1 (ya que tienen 1 o 3 bits en 1). Tenemos N números $B_i \in \{0, 1\}$, y queremos decidir si existe T tal que paridad($A_i \& T$) sea igual a B_i para todo $0 \leq i < N$.

Meet in the Middle

Veamos como venimos haciendo hasta ahora, la definición a través de un ejemplo.

Paridad

Tenemos N números A_i enteros entre 0 y $2^M - 1$ inclusive. Definimos la función paridad(X) como la cantidad de bits en 1 de X módulo 2. Por ejemplo: 0, 3, 5, 6 y 9 tienen paridad 0 (ya que tienen 0 o 2 bits en 1), mientras que 1, 2, 4, 7 y 8 tienen paridad 1 (ya que tienen 1 o 3 bits en 1). Tenemos N números $B_i \in \{0, 1\}$, y queremos decidir si existe T tal que paridad($A_i \& T$) sea igual a B_i para todo $0 \leq i < N$.

Veamos como resolveríamos esto con un backtracking clásico.

Paridad

```
bool esSolucion(int[] A, int[] B, int N, int mask):
    for i: 0 -> N
        if paridad(A[i] & mask) != B[i]
            return false
    return true
bool existeT(int[] A, int[] B, int N, int M):
    for i: 0 -> (1<<M)
        if esSolucion(A,B,N,i)
            return true
    return false
```

Paridad

```
bool esSolucion(int[] A, int[] B, int N, int mask):
    for i: 0 -> N
        if paridad(A[i] & mask) != B[i]
            return false
    return true
bool existeT(int[] A, int[] B, int N, int M):
    for i: 0 -> (1<<M)
        if esSolucion(A,B,N,i)
            return true
    return false
```

Este algoritmo tiene una complejidad de $\mathcal{O}(N \times 2^M)$ ya que hay 2^M valores para los cuales calculamos esSolucion, cuya complejidad es $\mathcal{O}(N)$.

Paridad con Meet in the Middle

```
int calcular(int[] A, int[] B, int N, int mask):
    int ret = 0
    for i: 0 -> N
        if paridad(A[i] & mask) != B[i]
            ret = (ret | (1<<i))
    return ret
int[] soluciones(int[] A, int[] B, int N, int M):
    int[1<<M] ret
    for i: 0 -> (1<<M)
        ret[i] = calcular(A,B,N,i)
    return ret
```

La función calcular devuelve un entero de N bits con 1 en las posiciones que representan los números para los cuales "falla" la paridad usando la máscara *mask*
La función soluciones devuelve este número para cada máscara posible y tiene una complejidad de $\mathcal{O}(N \times 2^M)$, veamos como usarla para bajar la complejidad de existeT.

Paridad con Meet in the Middle

```
bool existeT(int[] A, int[] B, int N, int M):
    int[] izq = soluciones(A[0..N)[0..M/2),[0..0],N,M)
    int[] der = soluciones(A[0..N)[M/2..M),B,N,M-M/2)
    ordenar(izq),ordenar(der)
    int j = 0
    for i: 0 -> size(izq)
        while j < size(der)-1 and der[j] < izq[i]
            j++
        if der[j] == izq[i]
            return true
    return false
```

Este código llama a soluciones dos veces con instancias de tamaño $\frac{M}{2}$, luego la complejidad en cada una es $\mathcal{O}(N \times 2^{M/2})$. Cada arreglo tiene tamaño $\mathcal{O}(2^{M/2})$ por lo que ordenar toma $\mathcal{O}(M \times 2^{M/2})$, luego el for es lineal por lo que la complejidad de existeT pasó de $\mathcal{O}(N \times 2^M)$ a $\mathcal{O}((N+M)2^{M/2})$

A ver, a ver... ¿y porqué anda esto?

Bajamos la complejidad prácticamente a su raiz cuadrada (un montón!), pero... ¿porqué anda? ¿Porqué pudimos hacer esto?

A ver, a ver... ¿y porqué anda esto?

Bajamos la complejidad prácticamente a su raiz cuadrada (un montón!), pero... ¿porqué anda? ¿Porqué pudimos hacer esto?

Para cada mitad calculamos para todas las posibles máscaras en qué números fallan. Elegir los primeros $\frac{M}{2}$ bits es independiente de lo que hagamos con los restantes $M - \frac{M}{2}$ bits, por lo que podemos elegirlos por separado, luego vamos a querer que los que fallaron a la izquierda, sean exactamente los que fallaron a la derecha, para "arreglar" estos errores, y por lo tanto, queremos ver si *izq* y *der* tienen algún número en común.

A ver, a ver... ¿y porqué anda esto?

Bajamos la complejidad prácticamente a su raiz cuadrada (un montón!), pero... ¿porqué anda? ¿Porqué pudimos hacer esto?

Para cada mitad calculamos para todas las posibles máscaras en qué números fallan. Elegir los primeros $\frac{M}{2}$ bits es independiente de lo que hagamos con los restantes $M - \frac{M}{2}$ bits, por lo que podemos elegirlos por separado, luego vamos a querer que los que fallaron a la izquierda, sean exactamente los que fallaron a la derecha, para "arreglar" estos errores, y por lo tanto, queremos ver si *izq* y *der* tienen algún número en común.

Ordenar ambos arreglos y movernos con dos punteros en $\mathcal{O}(N)$ es una de las muchas formas de buscar si dos arreglos tienen un número en común (ordenar uno y hacer búsqueda binaria para cada uno del otro, poner todos los de un arreglo en un conjunto y preguntar por los del otro, etc). Todas estas formas tienen complejidad $\mathcal{O}(N \log N)$

Bibliografía

- Programación Dinámica: https://goo.gl/Zn7X5t
- Máscaras de bits: https://goo.gl/WkQarR
- Técnicas algorítmicas en general: Capítulo 3 de http://goo.gl/VoXD9r[1]
- No podía faltar: Introduction to Algorithms (CLRS)[2]

[1] El libro es muy bueno, no sólo en lo que son técnicas algorítmicas, pero usa C++ para todo y abusa entre otras cosas de los define y typedef que muestran al principio del libro, lo que hace que el código sea un poco molesto para leer.

[2] La C es de Cormen.

Con ustedes... el TP1