Backtracking y Programación Dinámica

Víctor Racsó Galván Oyola

14 de mayo de 2021

Índice

Backtracking

Concepto

Idea

Construir todas las posibilidades para resolver un problema (como en Complete search) con la habilidad de deshacer partes de la construcción (backtrack). Se plantea con recursión.

Concepto

Idea

Construir todas las posibilidades para resolver un problema (como en Complete search) con la habilidad de deshacer partes de la construcción (backtrack). Se plantea con recursión.

Ventajas

Podemos ir descartando soluciones subóptimas o inválidas a medida que avanzamos en la construcción, reduciendo el espacio de búsqueda.

Concepto

Idea

Construir todas las posibilidades para resolver un problema (como en Complete search) con la habilidad de deshacer partes de la construcción (backtrack). Se plantea con recursión.

Ventajas

Podemos ir descartando soluciones subóptimas o inválidas a medida que avanzamos en la construcción, reduciendo el espacio de búsqueda.

Desventajas

En el peor de los casos, su redimiento es igual al de Complete search (no se descarta nada) con un recargo de tiempo por las llamadas recursivas.

Problema de muestra

Problema de las 8 reinas

Dado un tablero de ajedrez, contar la cantidad de formas en las que se pueden colocar 8 reinas bajo la condición de que, para cada par de reinas, ninguna amenace a otra.

Problema de muestra

Problema de las 8 reinas

Dado un tablero de ajedrez, contar la cantidad de formas en las que se pueden colocar 8 reinas bajo la condición de que, para cada par de reinas, ninguna amenace a otra.

Solución I

Probar una asignación directa con todas las posiciones del tablero y verificar si la configuración es válida una vez construida.
Complejidad: Orden de $\binom{64}{8} \cdot 8^2$.

Problema de muestra

Problema de las 8 reinas

Dado un tablero de ajedrez, contar la cantidad de formas en las que se pueden colocar 8 reinas bajo la condición de que, para cada par de reinas, ninguna amenace a otra.

Solución I

Probar una asignación directa con todas las posiciones del tablero y verificar si la configuración es válida una vez construida.
Complejidad: Orden de $\binom{64}{8} \cdot 8^2$.

Solución II

Notando que las reinas deben estar en diferentes columnas, nos basta probar todas las posibilidades de asignación por columna, descartando configuraciones que no son válidas sobre la marcha.
Complejidad: Orden de 8^{10}.

Problema de muestra

Problema de las 8 reinas

Dado un tablero de ajedrez, contar la cantidad de formas en las que se pueden colocar 8 reinas bajo la condición de que, para cada par de reinas, ninguna amenace a otra.

Solución I

Probar una asignación directa con todas las posiciones del tablero y verificar si la configuración es válida una vez construida.
Complejidad: Orden de $\binom{64}{8} \cdot 8^2$.

Solución II

Notando que las reinas deben estar en diferentes columnas, nos basta probar todas las posibilidades de asignación por columna, descartando configuraciones que no son válidas sobre la marcha.
Complejidad: Orden de 8^{10}.

Bonus

¿Cuál es la complejidad para n reinas en un tablero de $n \times n$?

Intentemos un problema en Aizu Online Judge

Exhaustive Search

Dada una secuencia A de longitud n y un entero M, determinar si existe algún subconjunto de A tal que la suma de dichos elementos sea M.

Intentemos un problema en Aizu Online Judge

Exhaustive Search

Dada una secuencia A de longitud n y un entero M, determinar si existe algún subconjunto de A tal que la suma de dichos elementos sea M.

Solución I - Pensemos el caso $q = 1$

Podemos usar recursión y definir la función $f(i, s)$, que nos responderá a la siguiente pregunta:
¿Se puede obtener un subconjunto con suma igual a M dado que ya he procesado los primeros i elementos y tomado un subconjunto con suma s?

Intentemos un problema en Aizu Online Judge

Exhaustive Search

Dada una secuencia A de longitud n y un entero M, determinar si existe algún subconjunto de A tal que la suma de dichos elementos sea M.

Solución I - Pensemos el caso $q = 1$

Podemos usar recursión y definir la función $f(i, s)$, que nos responderá a la siguiente pregunta:
¿Se puede obtener un subconjunto con suma igual a M dado que ya he procesado los primeros i elementos y tomado un subconjunto con suma s?

Solución I - Análisis (I)

Tenemos dos opciones para el elemento i:

- Agregarlo a nuestro subconjunto, en cuyo caso debemos sumar a_i a s para la siguiente transición. $f(i, s) \rightarrow f(i + 1, s + a_i)$.
- Ignorarlo, en cuyo caso el s se mantendrá como llegó. $f(i, s) \rightarrow f(i + 1, s)$.

Intentemos un problema en Aizu Online Judge - II

Ahora debemos definir dos cosas más: los casos base y cómo obtendremos la respuesta con esta función.

Intentemos un problema en Aizu Online Judge - II

Ahora debemos definir dos cosas más: los casos base y cómo obtendremos la respuesta con esta función.

Solución I - Análisis (II)

- Casos base: Cuando ya hemos procesado los primeros n elementos, no nos queda nada más por procesar, así que cuando tenemos $f(n, s)$ solo debemos verificar si $s = M$ para devolver `true` o `false` si no se cumple la igualdad.
- Antes de empezar a procesar los elementos, tenemos un subconjunto vacío; por lo tanto, la cantidad de elementos procesados es 0 y la suma es 0. Obtendremos la respuesta llamando a $f(0, 0)$.

Complejidad: $O(q \cdot 2^n)$. Veredicto: TLE

Intentemos un problema en Aizu Online Judge - II

Ahora debemos definir dos cosas más: los casos base y cómo obtendremos la respuesta con esta función.

Solución I - Análisis (II)

- Casos base: Cuando ya hemos procesado los primeros n elementos, no nos queda nada más por procesar, así que cuando tenemos $f(n, s)$ solo debemos verificar si $s = M$ para devolver `true` o `false` si no se cumple la igualdad.
- Antes de empezar a procesar los elementos, tenemos un subconjunto vacío; por lo tanto, la cantidad de elementos procesados es 0 y la suma es 0. Obtendremos la respuesta llamando a $f(0, 0)$.

Complejidad: $O(q \cdot 2^n)$. Veredicto: TLE

Solución II - ¿Y si consideramos f como preprocesamiento?

Podemos modificar la función f para que sea una función de tipo `void`, de manera que cuando lleguemos a $f(n, s)$ simplemente marcaremos que s es una suma que sí se puede formar (podemos hacer esto en un arreglo de booleanos global o en un set global). Con este preprocesamiento las consultas serán respondidas en $O(1)$ o $O(n)$ dependiendo de la implementación.
Complejidad: $O(n(2^n + q))$ si se usa set o $O(2^n + q)$ si se usa un arreglo de booleanos.

Programación Dinámica (DP)

¿Cómo resolvemos el problema anterior si $n \leqslant 50$?

Consideremos el mismo problema del Aizu pero esta vez aumentaremos los límites de n, de manera que una 2^n se demoraría unos cuantos años en terminar. ¿Qué haríamos en tal caso?

¿Cómo resolvemos el problema anterior si $n \leqslant 50$?

Consideremos el mismo problema del Aizu pero esta vez aumentaremos los límites de n, de manera que una 2^n se demoraría unos cuantos años en terminar. ¿Qué haríamos en tal caso?

Concepto

No reinventes la rueda, si algo ya está procesado, guárdalo para ya no tener que volverlo a procesar.

¿Cómo resolvemos el problema anterior si $n \leqslant 50$?

Consideremos el mismo problema del Aizu pero esta vez aumentaremos los límites de n, de manera que una 2^n se demoraría unos cuantos años en terminar. ¿Qué haríamos en tal caso?

Concepto

No reinventes la rueda, si algo ya está procesado, guárdalo para ya no tener que volverlo a procesar.

Transformación de backtracking a DP

Si los argumentos de la función de backtracking que hemos diseñado son fácilmente mapeables (Cualquier tipo que tenga definida el operador $<$), podemos usar una tabla (o map) de booleanos que nos diga si ya hemos procesado ese "estado" o situación, así como usaremos otra estructura similar para almacenar la respuesta que calculamos.

Transformando el problema del Aizu a DP - I

Originalmente, la solución con backtracking sería algo así:

```
const int MAX = 2000 + 5;

int n;
int q;
vector<int> a;
bool posible[MAX];

void f(int i, int s){
    if(s >= MAX) return;
        if(i == n){
                posible[s] = true;
                return;
        }
        f(i + 1, s + a[i]);
        f(i + 1, s);
}
```

Transformando el problema del Aizu a DP - II

Si le agregamos una tabla extra

```
const int N = 50 + 5;
const int MAX = 2000 + 5;

int n;
int q;
vector<int> a;
bool posible[MAX];
bool vis[N][MAX];

void f(int i, int s){
    if(s >= MAX) return;
        if(i == n){
                posible[s] = true;
                return;
        }
        if(vis[i][s]) return;
        f(i + 1, s + a[i]);
        f(i + 1, s);
}
```

Transformando el problema del Aizu a DP - III

La complejidad se reduce a $O(n \cdot MAX)$, pues en el peor de los casos rellenaremos toda la tabla *vis* y después de eso siempre haremos `return`.
Esta complejidad nos permite tener el n mucho más grande que antes.

Reconstruyendo la solución

Para reconstruir una solución con DP nos basta con definir una tabla extra de las mismas dimensiones que *vis* que nos especifique qué decisión hemos tomado para llegar a la respuesta.

Versión iterativa

Podemos intentar buscar la forma correcta de rellenar las tablas de DP sin necesidad de usar recursión, pero esto requiere un análisis un poco más detallado de las dependencias (qué estado debe tener su respuesta asignada antes que el que deseamos procesar).

Problemas clásicos con DP

Educational DP Contest - AtCoder

Hay muchos problemas clásicos de Programación Dinámica, desde nivel básico hasta avanzado, es una buena práctica intentar resolver todos.

Uhunt - Capítulo 3

Los problemas del Uhunt están para complementar el libro Competitive Programming 3, así que su selección de problemas para Programación Dinámica también es una fuente valiosa de entrenamiento.