Complete Search

Por Ariel Parra

[ΓΩ5]

Definicion

La búsqueda exhaustiva (Complete Search) es un método que utiliza fuerza bruta (Brute Force) para generar todas las soluciones posibles a un problema y seleccionar la mejor o contar el número de soluciones. Es útil cuando se dispone de suficiente tiempo, ya que es fácil de implementar y garantiza la respuesta correcta. Si resulta demasiado lenta, se pueden emplear técnicas como algoritmos voraces (Greedy) o programación dinámica (DP).

Permutations

Un método para generar permutaciones consiste en comenzar con la permutación $\{ 0, 1, \ldots, n - 1 \}$ y utilizar repetidamente una función que construya la siguiente permutación en orden creciente.

```
vector<int> permutation;
for (int i = 0; i < n; i++) {
    permutation.push_back(i);
}
do {
    // procesar permutación
} while (next_permutation(permutation.begin(), permutation.end()));
```

Subsets

Primero consideramos el problema de generar todos los subconjuntos de un conjunto de n elementos. Para por ejemplo, los subconjuntos de {0, 1, 2} son ∅={}, {0}, {1}, {2}, {0, 1}, {0, 2}, {1, 2} y {0, 1, 2}.

Una forma de generar subconjuntos se basa en la representación en bits de los números enteros. Cada subconjunto de un conjunto de n elementos se puede representar como una secuencia de n bits, lo que corresponde a un entero entre 0 y 2^n - 1. Los unos en la secuencia de bits indican qué elementos están incluidos en el subconjunto.

La convención usual es que el último bit corresponde al elemento 0, el penúltimo al elemento 1, y así sucesivamente. Por ejemplo, la representación en bits del número 25 es 11001, lo que corresponde al subconjunto { 0, 3, 4 } .

El siguiente código recorre los subconjuntos de un conjunto de n elementos:

```
for (int b = 0; b < (1 << n); b++)
    // procesar subconjunto
```

El código a continuación muestra cómo podemos encontrar los elementos de un subconjunto que corresponde a una secuencia de bits. Al procesar cada subconjunto, el código construye un vector que contiene los elementos en el subconjunto.

```
for (int b = 0; b < (1 << n); b++) {
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b & (1 << i)) subset.push_back(i);
    }
}
```

Meet in the middle

La técnica **meet in the middle** (o "encuentro en el medio") divide el espacio de búsqueda en dos partes de tamaño aproximadamente igual. Se realiza una búsqueda separada en cada una de estas partes y, finalmente, se combinan los resultados.Esta técnica es útil si se puede combinar los resultados de las búsquedas de manera eficiente.

Ejemplo

Supongamos un problema en el que se tiene una lista de *n* números y un número *x*, y se desea saber si es posible elegir algunos números de la lista de modo que su suma sea *x*. Por ejemplo, con la lista $[2, 4, 5, 9]$ y $x = 15$, podemos seleccionar los números $[2, 4, 9]$ para obtener $2 + 4 + 9 = 15$. Sin embargo, si $x = 10$ con la misma lista, no es posible formar dicha suma.

Un algoritmo simple para resolver este problema es recorrer todos los subconjuntos de los elementos y verificar si la suma de alguno de ellos es *x*. Este enfoque tiene una complejidad de $O(2^n)$, ya que existen 2^n subconjuntos. Sin embargo, usando la técnica **meet in the middle**, se puede reducir la complejidad a $O(2^{n/2})$.

Algoritmo de solucion al ejemplo

La idea es dividir la lista original en dos listas, *A* y *B*, de modo que ambas contengan aproximadamente la mitad de los números. La primera búsqueda genera todos los subconjuntos de *A* y almacena sus sumas en una lista SA. De forma similar, la segunda búsqueda crea una lista SB con las sumas de los subconjuntos de *B*. Para encontrar una combinación que dé la suma *x*, basta con verificar si es posible elegir un elemento de SA y otro de SB cuya suma sea igual a *x*.

Por ejemplo, con la lista $[2, 4, 5, 9]$ y $x = 15$:

1. Dividimos la lista en $A = [2, 4]$ y $B = [5, 9]$.
2. Generamos las listas $SA = [0, 2, 4, 6]$ y $SB = [0, 5, 9, 14]$.
3. Notamos que la suma $x = 15$ se puede formar, ya que en SA se encuentra la suma 6, en SB la suma 9, y $6 + 9 = 15$, lo que corresponde a la solución $[2, 4, 9]$.

Algoritmo que calcula la mayor suma posible de un subconjunto que sea menor o igual a un valor dado S

```
ll X[2000005],Y[2000005];// global vectors
// Find all possible sum of elements of a[] and store in x[]
void calcsubarray(vector<ll> a, int n, int c) {
    for (int i=0; i<(1<<n); ++i) {
        ll s = 0;
        for (int j=0; j<n; ++j)  if (i & (1<<j)) s += a[j+c];
        x[i] = s;
    }
}
```

```
int main() {
    vector<ll> a = {3, 34, 4, 12, 5, 2};
    int n=a.size();
    ll S = 10;
    cout<<"Largest value smaller than or equal to given sum is"<< solveSubsetSum(a,n,S);
}
```

```
ll solveSubsetSum(vector<ll> a, int n, ll S) {
    // Compute all subset sums of first and second halves
    calcsubarray(a, X, n/2, 0);    calcsubarray(a, Y, n-n/2, n/2);
    int size_X = 1<<(n/2);    int size_Y = 1<<(n-n/2);
    // Sort Y (we need to do doing binary search in it)
    sort(Y, Y+size_Y);
    // To keep track of the maximum sum of a subset such that the maximum sum is less than S
    ll max = 0;
    // Traverse all elements of X and do Binary Search for a pair in Y with maximum sum less than S.
    for (int i=0; i<size_X; ++i) {
        if (X[i] <= S) {
            // lower_bound() returns the first address which has value greater than or equal to S-X[i].
            int p = lower_bound(Y, Y+size_Y, S-X[i]) - Y;
            // If S-X[i] was not in array Y then decrease p by 1
            if (p == size_Y || Y[p] != (S-X[i]))    p--;
            if ((Y[p]+X[i]) > max)  max = Y[p]+X[i];
        }
    }
    return max;
}// O(2^(n/2))
```

Problema

- **102951A** Maximum Distance ⮭

Referencias

- Laaksonen, A. (2018). *Competitive Programmer's Handbook*. Recuperado de https://cses.fi/book/book.pdf
- Many . (s.f). *Complete Search with Recursion*. Recuperado de https://usaco.guide/bronze/complete-rec?lang=cpp
- Yao,D.& Liu, D. (s.f.). *Basic Complete Search*. Recuperado de https://usaco.guide/bronze/intro-complete?lang=cpp