Bits Mask & Next permutation

HeNeos

3 de febrero de 2019

1 PC-UNI

Objetivos

- Analizar problemas que requieran computar todas las permutaciones de un arreglo
- Analizar problemas que requieran computar todos los subconjuntos de un arreglo
- Mostrar como hacer sobrecarga de operadores

Problema motivacinal 1

Dado un entero n. Imprimir todas las permutaciones de $\{1,2,3,\ldots,n\}$. Límites: $0 \leq n \leq 9$.

Una solución a este problema es simplemente usar la STL. Pues, hay funciones que nos pueden ayudar a hacer esta tarea:

- `next_permutation`
- `iota`

Obteniendo una solución $O(n \cdot n!)$

Programa 1: Problema motivacional 1

```cpp
#include <bits/stdc++.h>

#define all(X) begin(X), end(X)

using namespace std;

int main () {
  int n;
  cin >> n;
  vector <int> p(n);
  iota(all(p), 1);
  do {
    for (int p_i: p) cout << p_i << ' ';
    cout << endl;
  } while(next_permutation(all(p)));
  return (0);
}
//Codigo hecho con style=manni
```

Problema motivacional 2

Dado un entero n seguido de n enteros: $a_1, a_2, a_3, \ldots, a_n$. Imprimir la suma de elementos de cada subconjunto de los n dados. Límites: $1 \leq n \leq 20$.

Una solución a este problema es usando máscara de bits. Para entender ello, primero veamos algunos datos útiles:

- Un `int` usa 32-bit
- Un `long long` usa 64-bit
- Un entero que usa k bits puede almacenar números en $[-2^{k-1}, 2^{k-1})$
- Un entero sin singo (`unsigned`) que usa k bits puede almacenar números en $[0, 2^k)$

En C++ podemos comprobar lo anterior con el siguiente código:

Ejemplo 2: Comprobación

```
cout << INT_MIN << ' ' << INT_MAX << endl;
cout << UINT_MAX << endl;
cout << LLONG_MIN << ' ' << LLONG_MAX << endl;
cout << ULLONG_MAX << endl;
//Codigo hecho con style=igor
```

Los enteros se guardan en su representación binaria usando k bits (completando con 0's si es necesario).
Por ejemplo, para un entero de 8 bits, los siguientes números se guardarían así:

- 5=00000101
- 7=00000111
- 8=00001000
- 12=00001100
- 19=00010011

En C++ podemos comprobar lo anterior con el siguiente código:

Ejemplo 3: Comprobación

```
for (int num: {5, 7, 8, 12, 19}) {
    cout << setw(2) << num << ' ' << bitset <8>(num) << endl;
  }
//Codigo hecho con style=lovelace
```

En C++ podemos trabajar con los números a nivel de bits usando estas operaciones:

- &
- |
- $\wedge$
- $\sim$

Además en C++ podemos mover todos los bits de un número hacia la izquierda o la derecha con los operadores << y >> respectivamente.
Con esto, podemos:

Obtener 2^k: `1 << k`

Alternar el k-esimo bit: `x` $\wedge$ `(1 << k)`

Apagar el k-esimo bit: `x & (`$\sim$`(1 << k))`

Prender el k-esimo bit: `x | (1 << k)`

Obtener el k-esimo bit: `(x >> k) & 1`

Así, podemos resolver nuestro problema motivacional en $O(n \cdot 2^n)$ con el siguiente código:

Programa 4: Problema motivacional 2

```
int n;
  cin >> n;
  vector <int> arr(n);
  for (int i = 0; i < n; i++) cin >> arr[i];
  for (int mask = 0; mask < (1 << n); mask++) {
    int sum = 0;
    for (int bit = 0; bit < n; bit++) {
      if ((mask >> bit) & 1) sum += arr[bit];
    }
    cout << bitset <20> (mask) << " suma = " << sum << endl;
  }
//Codigo hecho con style=xcode
```

Lo anterior también te permitiría resolver (con un poco de ingenio) algunas problemas que requieran manipular bits.
También te podría interesar:

- `_builtin_popcount`
- `_builtin_clz`
- `_builtin_ctz`
- ¿Cómo se guardan los números negativos?
- La clase de manejo de bits del año pasado

Problema motivacional 3

Dado un entero n seguido de n pares de elementos a_i, b_i (que representan a la fracción a_i/b_i). Imprimir las n fracciones en forma creciente. Límites: $1 \leq n \leq 10^5$.

Iremos explicando en clase como llegar a codear esto:

Programa 5: Prueba

```cpp
/**
 * Ejemplo de como crear un 'struct' y como usarlo
 * Problema
 * - Queremos tener un tipo de dato de represente una fraccion
 * - Queremos poder ordenar fracciones siguiendo esta relacion de orden
 *      a1   a2
 *      -- < --      <->     a1 * b2 < a2 * b1
 *      b1   b2
 * - Queremos poder usar la operacion '+' y '*' entre fracciones
 * Recibire un numero 'n' seguido de 'n' fracciones
 * Debo imprimir
 * Las fracciones ordenadas
 * La suma de las fracciones
 * El producto de las fracciones
 **/
// Para simplificar la solucion
// aceptare que las fracciones no tienen 0 en el denominador
#include <bits/stdc++.h>
using namespace std;
struct Fraccion {
    int num, den;
    // Los constructores tienen el mismo nombre que mi estructura
    // Puedo tener mas de un constructor
    // Estos se diferenciaran por los parametros que reciban
    Fraccion() {}
    Fraccion(int x, int y) {
        num = x;
        den = y;
    }
    // Puedo sobreescribir el operador '<' para poder usar la funcion 'sort'
    // - (Fraccion& otra)
    //   Indica que esta funcion recibira una referencia de la variable que invoque
    //   este metodo. Asi, no se creara una copia de esa variable para esta funcion
    //   Lo que se haga con esta variable dentro de la funcion se vera reflejado en
    //   la variable que la invoco
    // - (const Fraccion otra)
    //   Indica que el valor de la variable 'otra' sera constante en ese metodo
    // - () const {
    //   Indica que este metodo no cambiara el estado de ningun atributo de
    //   la instancia de esta 'struct' que invoque el metodo
    bool operator < (const Fraccion& otra) const {
        return num * otra.den < den * otra.num;
    }
    // Si deseo, puedo no usar 'const' y '&'
    // Pero esto sera un poco mas lento ya que crea copias innecesarias
```

```
46      // Y al no definir que es un 'const', no da libertad al compilador de
47      // hacer optimizaciones
48      Fraccion operator + (Fraccion otra) {
49          return Fraccion(num * otra.den + den * otra.num, den * otra.den);
50      }
51      // Este operador cambiara el estado de la instancia que la invoque
52      // por ello no podemos usar () const {
53      void operator *= (const Fraccion& otra){
54          num = num * otra.num;
55          den = den * otra.den;
56      }
57      // Tambien puedo escribir funciones ('metodos') para este 'struct'
58      void imprimir(string sep) {
59          cout << num << '/' << den << sep;
60      }
61  }; // NO OLVIDAR PONER ';' al final de un 'struct'
62  int n;
63  vector <Fraccion> arr;
64  // Si no deseo sobreescribir el operador ('<') en mi 'struct', puedo definirlo
65  // como una funcion, asi:
66  bool cmp(const Fraccion& X, const Fraccion& Y) {
67      return X.num * Y.den < X.den * Y.num;
68  }
69  int main() {
70      cin >> n;
71      for (int i = 0; i < n; i++) {
72          int num, den;
73          cin >> num >> den;
74          arr.push_back(Fraccion(num, den));
75      }
76      sort(arr.begin(), arr.end());
77      //sort(arr.begin(), arr.end(), cmp);
78      cout << "Las fracciones ordenadas" << endl;
79      for (auto fraccion : arr) fraccion.imprimir("  ");
80      cout << endl;
81      Fraccion suma = Fraccion(0, 1);
82      for (auto fraccion : arr) {
83          suma = suma + fraccion;
84          // Si quisiera usar suma += fraccion
85          // Tendria que definir el operador '+='
86      }
87      cout << "La suma de las fracciones es" << endl;
88      suma.imprimir("\n");
89      Fraccion producto = Fraccion(1, 1);
90      for (auto fraccion : arr) producto *= fraccion;
91      cout << "El producto de las fracciones es" << endl;
92      producto.imprimir("\n");
93
94      return (0);
95  }
96  //Codigo hecho con style=vs
```

Te podría interesar investigar sobre: `stable_sort`

2 Investigación

`next_permutation`

Se utiliza para reorganizar los elementos en el rango `[first, last)` en la siguiente permutación lexicográficamente mayor. La función es del tipo `bool`, por lo que solo devuelve `true` o `false`.

Ejemplo 6: next_permutation

```cpp
#include <algorithm>

#include <iostream>

using namespace std;

int main() {
    int arr[] = { 1, 2, 3 };
    sort(arr, arr + 3);

    cout << "The 3! possible permutations with 3 elements:\n";
    do {
        cout << arr[0] << " " << arr[1] << " " << arr[2] << "\n";
    } while (next_permutation(arr, arr + 3));

    cout << "After loop: " << arr[0] << ' '
         << arr[1] << ' ' << arr[2] << '\n';

    return 0;
}
//Salida:
//The 3! possible permutations with 3 elements:
//1 2 3
//1 3 2
//2 1 3
//2 3 1
//3 1 2
//3 2 1
//After loop: 1 2 3

//Codigo hecho con style=autumn
```

`prev_permutation`

De la misma forma que `next_permutation` solo que devuelve una permutación anterior.

Ejemplo 7: prev_permutation

```cpp
#include <algorithm>

#include <iostream>

using namespace std;

int main() {
    int arr[] = { 1, 2, 3 };
    sort(arr, arr + 3);
    reverse(arr, arr + 3);

    cout << "The 3! possible permutations with 3 elements:\n";
    do {
        cout << arr[0] << " " << arr[1] << " " << arr[2] << "\n";
    } while (prev_permutation(arr, arr + 3));

    cout << "After loop: " << arr[0] << ' ' << arr[1]
         << ' ' << arr[2] << '\n';

    return 0;
}
//Salida:
//The 3! possible permutations with 3 elements:
//3 2 1
//3 1 2
//2 3 1
//2 1 3
//1 3 2
//1 2 3
//After loop: 3 2 1

//Codigo hecho con style=abap
```

`iota`

Asigna a cada elemento en el rango `[first,last)` valores sucesivos de `val`, como si se tratará de `val++`.

Ejemplo 8: iota

```cpp
#include <iostream>
#include <numeric>
using namespace std;

int main() {
    int numbers[10];
    // Initailising starting value as 100
    int st = 100;

    iota(numbers, numbers + 10, st);

    cout << "Elements are :";
    for (auto i : numbers)
        cout << ' ' << i;
    cout << '\n';

    return 0;
}
//Salida:
//Elements are : 10 11 12 13 14 15 16 17 18 19 20

//Codigo hecho con style=rrt
```

`bitset`

Un `bitset` es un `array of bool`, pero cada valor booleano no se almacena por separado, en lugar de eso, `bitset` optimiza el espacio de tal manera que cada booleano toma 1 espacio de bits, por lo que el espacio que toma el `bitset` es menor de lo que tomaría un `array`. Podemos acceder a cada bit del `bitset` con la ayuda del operador de indexación de un `array,'[]'`, entonces `array[3]` muestra el bit en el índice 3 del `bitset`, recordando que se lee de derecha a izquierda y comienza en 0.

Ejemplo 9: bitset

```cpp
#include <bits/stdc++.h>
using namespace std;
#define M 32
int main() {
    bitset<M> bset1;     // Constructor por default, establece todos los bits en 0
    bitset<M> bset2(20); // bset2 es inicializado con los bits de 20
    bitset<M> bset3(string("1100"));// bset3 es inicializado con los bits del string
    cout << bset1 << endl; // 00000000000000000000000000000000
    cout << bset2 << endl; // 00000000000000000000000000010100
    cout << bset3 << endl; // 00000000000000000000000000001100
    // declarando set8 con capacidad de 8 bits
    bitset<8> set8; // 00000000
    set8[1] = 1; // 00000010
    set8[4] = set8[1]; // 00010010
    int numberof1 = set8.count(); // count retorna el numero de 1's en el bitset

    // size function retorna el numero total de bits en el bitset.
    int numberof0 = set8.size() - numberof1;
    cout << set8 << " has " << numberof1 << " ones and "
         << numberof0 << " zeros\n";

    // test function retorna 1 si el bit esta establecido, sino 0
    cout << "bool representation of " << set8 << " : ";

    for (int i = 0; i < set8.size(); i++) cout << set8.test(i) << " ";
    // any function retorna true, si al menos 1 bit esta establecido.
    if (!set8.any()) cout << "set8 has no bit set.\n";
    // none function retorna true, si no hay ningun bit establecido.
    if (!bset1.none()) cout << "bset1 has some bit set\n";

    cout << set8.set() << endl; //1 en todos los bits
    cout << set8.set(4, 0) << endl; //0 en el indice 4
    cout << set8.set(4) << endl; //1 en el indice 4
    cout << set8.reset(2) << endl; //0 en el indice 2
    cout << set8.reset() << endl;  //0 en todo los bits
    cout << set8.flip(2) << endl; //flip en el indice 2
    cout << set8.flip() << endl; //flip en todos los bits
    // Converting decimal number to binary by using bitset
    int num = 100;
    cout << "\nDecimal number: " << num
         << "  Binary equivalent: " << bitset<8>(num);
    return 0;
}
//Codigo hecho con style=perldoc
```

__builtin

__builtin_popcount

Esta función es usada para contar el número de 1's de un número:

Ejemplo 10: popcount

```
#include <stdio.h>
int main() {
    int n = 5;
    printf("Count of 1s in binary of %d is %d ",n, __builtin_popcount(n));
    //Count of 1s in binary of 5 is 2
}
//Codigo hecho con style=arduino
```

__builtin_parity

Ejemplo 11: popcount

```
#include <stdio.h>
int main() {
    int n = 7;
    printf("Parity of %d is %d ",n, __builtin_parity(n));
    //Parity of 7 is 1
}
//Codigo hecho con style=tango
```

__builtin_clz

Cuenta el número de 0's después del primer 1.

Ejemplo 12: popcount

```
int main() {
    int n = 16;
    printf("Count of leading zeros before 1 in %d is %d",n, __builtin_clz(n));
    //Count of leading zeros before 1 in 16 is 27
}
//Codigo hecho con style=emacs
```

__builtin_ctz

Cuenta el número de 0's hasta el primer 1.

Ejemplo 13: popcount

```
int main() {
    int n = 16;
    printf("Count zeros from last to first occurrence of one is %d",builtin_ctz(n));
    //Count zeros from last to first occurrence of one is 4
}
//Codigo hecho con style=friendly
```

`stable_sort`

Funciona igual que `sort`, solo que si encontrase elementos iguales al comparar; los muestra manteniendo el orden.

Ejemplo 14: bitset

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool compare_as_ints (double i,double j){
  return (int(i)<int(j));
}

int main () {
  double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
  vector<double> myvector;
  myvector.assign(mydoubles,mydoubles+8);
  cout << "using default comparison:";
  stable_sort (myvector.begin(), myvector.end());
  for (vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';
  myvector.assign(mydoubles,mydoubles+8);
  cout << "using 'compare_as_ints' :";
  stable_sort (myvector.begin(), myvector.end(), compare_as_ints);
  for (vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';
  return 0;
}
//Salida
//using default comparison: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
//using compare_as_ints: 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67

//Codigo hecho con style=borland
```