Entrenamiento 5: Técnicas de resolución de problemas

Input y output rápido

Para leer e imprimir datos de una forma mucho más rápida en contexto de competición, podemos incluir estas dos líneas al principio de `main`:

```
ios::sync_with_stdio(false);
cin.tie(NULL);
```

Cálculo de Potencias en O(log n)

Para calcular n^x, el método más sencillo es multiplicar n por sí mismo x veces. Sin embargo, es terriblemente ineficiente y no aprovechamos los resultados que ya conocemos. El método rápido consiste en aprovechar que $n^x = (n^{x/2})^2$. Si x es par, retornamos $n^{x/2}$ al cuadrado. Si x es impar, calculamos $(n^{(x-1)/2})^2$ y multiplicamos por n, retornando el resultado. Así:

```
int exponenciacion (int base, int potencia) {
    if (potencia == 0) return 1;
    int x = exponenciacion(base, potencia>>1);
    if (potencia & 1) return x*x*base;
    else return x*x;
}
```

(Los operadores >> y & son operadores con bits: formas más rápidas de operar con 2 y sus potencias. potencia >> 1 = potencia/2 y potencia & 1 = potencia % 2 == 1).

Breve Apunte de Aritmética Modular

Cuando un problema te pide una respuesta módulo un primo grande (suele ser 10^9+7), suele ser porque la respuesta al problema es tan grande para inputs grandes que no entra en un long long int. Es por esto que hay que tener mucho cuidado con tener un overflow. No vale con simplemente calcular la respuesta y al final del todo hacer módulo. Hay que ir haciendo módulo en pasos intermedios. Por este motivo es útil saber que:

- (a+b)%m = (a%m + b%m)%m
- Si a – b da un número negativo, (a-b)%m en C++ dará un resultado negativo. Para evitar que esto ocurra, conviene hacer ((a%m) – (b%m) + m)%m.
- (a*b)%m = ((a%m) * (b%m))%m.

Veremos más adelante que hay que tener cuidado juntando módulos y divisiones en el mismo problema, pero de momento no habrá problemas que requieran esto.

En resumen, lo fundamental es ir haciendo módulo 10^9+7 (o el primo en cuestión) cada poco para asegurarse de que no hay overflow.

Método de Bisección

Numerosos problemas te piden que encuentres el máximo/mínimo número *k* que cumple cierta característica (como *La Pachanga*, por ejemplo). En muchas ocasiones, es más sencillo comprobar si un número *kDePrueba* cumple esa característica que ir directamente a encontrar el mínimo/máximo.

En estos casos, usamos el método de bisección, donde dividimos el intervalo de búsqueda en dos con cada iteración hasta quedarnos con un intervalo de longitud 1, que será la respuesta. En ambas alternativas, la variable `lo` indica el número más pequeño del intervalo actual y `hi` el mayor número del intervalo actual.

```
bool esPosible(int k) {
    // aquí comprobamos si esa k funciona
}

// versión para buscar el mínimo
int minimoK() {
    int lo = 0, hi = 1e9, mid, k=1e9;
    while(lo <= hi) {
        mid = lo + (hi-lo)/2;
        if (esPosible(mid)) {
```

```
            hi = mid-1;
            k = mid;
        } else lo = mid+1;
    }
    return k;
}

// versión para buscar el máximo
int maximoK() {
    int lo = 0, hi = 1e9, mid, k=0;
    while(lo <= hi) {
        mid = lo + (hi-lo)/2;
        if (esPosible(mid)) {
            lo = mid+1;
            k = mid;
        } else hi = mid-1;
    }
    return k;
}
```

Combinatoria y Programación Dinámica

Es útil conocer las principales reglas de combinatoria, ya que pueden aparecer en algunos problemas:

- Regla del producto: si hay A elementos en un conjunto C_1 y B elementos en otro conjunto C_2, hay A*B formas de escoger un elemento de C_1 y otro de C_2.
- Regla de la suma: si hay |A| elementos en un conjunto A y |B| elementos en otro conjunto B, hay |A|+|B| formas de escoger un elemento de A o B, asumiendo que A y B no tienen elementos en común (un número de la unión de A y B).

- La fórmula de inclusión-exclusión:
  $|A \cup B| = |A|+|B|-|A \cap B|$
  El número de elementos en A o B es igual al número de elementos en A más el número de elementos en B menos el número de elementos en A y B. Es importante aplicar bien esta fórmula para evitar contar elementos dos veces.

Conviene conocer la principal terminología:

- Permutación: una manera de ordenar una serie de elementos. Por ejemplo, las permutaciones de la lista (1,2,3) son (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1).
- Combinación: una manera de seleccionar k elementos de un conjunto de n elementos. Por ejemplo, las combinaciones de la lista (1,2,3) de tamaño 2 son (1,2), (1,3) y (2,3).
- Subconjunto: un conjunto B es subconjunto de un conjunto A si todos los elementos de B están incluidos en A. Los subconjuntos del conjunto (1,2,3) son (), (1), (2), (3), (1,2), (1,3), (2,3), (1,2,3). Si el conjunto A tiene tamaño n, tiene 2^n subconjuntos. Los subconjuntos de un conjunto A son todas las combinaciones de todos los tamaños entre 0 y n del conjunto A.

El número de permutaciones de un conjunto A de tamaño n es igual a n!.

El número de combinaciones de tamaño k de un conjunto A de tamaño n es igual a n!/(k! * ((n-k)!)).

A la hora de programar, podemos encontrar n! = factorial(n) de una forma muy sencilla: factorial(n) = factorial(n-1)*n, con un caso base de factorial(0) igual a 1. Si vamos guardando los valores de factorial en un vector DP, podremos obtener repetidos factoriales de una manera mucho más eficiente.

```
int factorial(vector<int> & DP, int n) {
    if (DP[n] != -1)
        return DP[n];
    if (n == 0)
        return DP[0] = 1;
    return DP[n] = n * factorial(DP, n-1);
```

```
}
```

Para encontrar combinaciones, podemos hacer algo más eficiente y útil en el caso de números grandes y utilización de módulos que tratar de dividir los factoriales (recordemos que mezclar divisiones y módulos es trabajoso y veremos más adelante cómo hacerlo, pero en general conviene evitarlo si es posible). Usaremos la siguiente fórmula recursiva:

combinaciones(n, k) = combinaciones(n-1, k-1) + combinaciones(n-1, k).

```
int combinaciones(vector<vector<int>> & DP, int n, int k) {
      if (DP[n][k] != -1)
            return DP[n][k];
      if (n == k || k == 0)
            return DP[n][k] = 1;
      return DP[n][k] = combinaciones(DP, n-1, k-1) +
combinaciones(DP, n-1, k);
}
```

Algoritmos Voraces

Un algoritmo voraz es lo contrario a la programación dinámica. En vez de pensar a largo plazo, tomando decisiones medidas y teniendo en cuenta todo lo que ya sabe, el algoritmo elige en cada iteración lo que es óptimo en ese instante.

Por ejemplo, a la hora de devolver cambio, un dependiente suele pagar con el billete más alto pero menor a la cantidad restante y hace esto sucesivamente hasta que no falte cambio por devolver. Por cómo está diseñado el euro, con nuestra moneda esto es óptimo, pero no tiene por qué ser el caso.

Frecuentemente, se usan algoritmos voraces para encontrar soluciones rápidas, pero no necesariamente óptimas. Hoy vamos a concentrarnos en los algoritmos voraces que sí nos garantizan soluciones óptimas.

Estrategia ordena+voraz

La mayoría de problemas a resolver con la estrategia voraz tienen una solución sencilla que consiste en ordenar una lista (frecuentemente con comparador propio) y aplicar el algoritmo a la lista ya ordenada. Un ejemplo es el problema de la semana pasada *Las galletas de Jan*.

Cubriendo intervalos

Imagina que tienes una calle, que modelas como una línea recta en una dimensión, y hay una serie de farolas en varios puntos de la calle. Cada farola ilumina un radio *r*, no necesariamente el mismo que el resto de farolas. ¿Cuál es el mínimo número de farolas que tienes que encender para iluminar toda la recta?

Este es un ejemplo de problema del tipo *cubriendo intervalos*. Representaremos cada farola con una pareja de valores: la coordenada mínima y máxima que ilumina dicha farola. Una vez construida la lista de coordenadas, la ordenamos de menor a mayor. La solución consiste en ir de izquierda a derecha encendiendo la farola más cercana cuando nos falte luz.

Una variación de este problema es el tener que iluminar farolas para cubrir a cierto número de vecinos, en cuyo caso conviene ordenar a los vecinos de izquierda a derecha y procesar la lista de vecinos en orden, iluminando farolas nuevamente si hace falta.

Búsqueda Completa y Optimización

Subconjuntos

Para iterar sobre los subconjuntos de un vector `lista` hacemos backtracking posición a posición, primero con esa posición sin seleccionar y luego con ella seleccionada. Se para el backtracking una vez que el índice llega a ser igual a la longitud de la lista, lo que significa que ya hemos procesado todos los elementos de la lista y, por tanto, tenemos un subconjunto listo para lo que necesitemos.

```
void subconjuntos(vector<int> & subconjunto, vector<int> &
lista, int ind) {
      if (ind == (int) lista.size()) {
            // imprimimos el subconjunto
            for (int i = 0; i < (int) subconjunto.size(); i++)
                  cout << subconjunto[i] << ' ';
            cout << '\n';
            return;
      }
```

```
	// subconjuntos sin el elemento
	subconjuntos(subconjunto, lista, ind+1);

	// cogemos el elemento
	subconjunto.push_back(lista[ind]);
	subconjuntos(subconjunto, lista, ind+1);
	subconjunto.pop_back();
}
```

Permutaciones

Encontrar las permutaciones de una lista es muy similar a encontrar los subconjuntos. Hacemos backtracking, esta vez sobre el tamaño de la permutación. Empezamos con una permutación vacía y vamos añadiéndole un elemento de cada vez hasta que el tamaño de la permutación es el mismo que el de la lista y, por tanto, están en ella todos los elementos. Para evitar repetir elementos, tenemos un vector booleano que guarda para cada posición dentro de la lista si esta ha sido incluida ya o no. A través de un bucle, conseguimos que, para cada posición dentro de la permutación, todos los elementos de la lista sean usados.

```
void permutaciones(vector<int> & permutacion, vector<bool> &
seleccionado, vector<int> & lista) {
	if (permutacion.size() == lista.size()) {
		// hacer algo con la permutacion
		for (int i = 0; i < (int) permutacion.size(); i++)
			cout << permutacion[i] << ' ';
		cout << '\n';
		return;
	}

	for (int i = 0; i < (int) lista.size(); i++) {
		if (!seleccionado[i]) {
			seleccionado[i] = true;
```

```
                permutacion.push_back(lista[i]);
                permutaciones(permutacion, seleccionado, lista);
                permutacion.pop_back();
                seleccionado[i] = false;
            }
        }
    }
```