Evaluación Final Matemática Discreta I

Karlos Alejandro Alfonso Rodríguez
Correo: karlos.alfonso@estudiantes.matcom.uh.cu
Grupo: C-213

13 de julio de 2021

1. Problema 888D

- Link del Problema: https://codeforces.com/problemset/problem/888/D
- Link del submit (Accepted) en Codeforces: https://codeforces.com/contest/888/submission/121990220

1.1. Resumen en Términos Matemáticos

Dados dos enteros positivos n, k donde $4 \leq n \leq 1000$ y $1 \leq k \leq 4$, y sea el conjunto $\mathbb{N}_n = \{1, 2, ..., n\}$, llamémosle p_i al elemento $p \in \mathbb{N}_n$ y ocupa la posición i.
El problema consiste en contar la cantidad de permutaciones de $\mathbb{N}_n$ que cumplan que sus elementos $p_i = i$ para al menos $n-k$ índices i, o sea, la cantidad de permutaciones de $\mathbb{N}_n$ donde al menos $n-k$ elementos de en la permutación conserven su posición original con respecto a $\mathbb{N}_n$.
Es equivalente decir que solopodrán haber a lo sumo k índices i que cumplan $p_i \neq i$.
De ahora en adelante a los elementos $p_i \neq i$ los llamaremos *"mal ubicados"*.

1.2. Problema Original

D. Almost Identity Permutations
time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

A permutation p of size n is an array such that every integer from 1 to n occurs exactly once in this array.
Let's call a permutation an *almost identity permutation* if there exist at least $n-k$ índices i $(1 \leq i \leq n)$ such that $p_i = i$.
Your task is to count the number of *almost identity* permutations for given numbers n and k.

Input
The first line contains two integers n and k $(4 \leq n \leq 1000, 1 \leq k \leq 4)$.

Output
Print the number of *almost identity* permutations for given n and k.

Examples

input	output
4 1	1
4 2	7
5 3	31
5 4	76

1.3. Solución Teórica del Problema

Para resolver este problema enfoquémonos en el hecho de que una permutación será casi la identidad si tiene a lo sumo k elementos *"mal ubicados"*.
Sea un entero m tal que $0 \leq m \leq k$. Dado un m fijo veamos si podemos hallar cuantas permutaciones son casi la identidad.

Primero debemos saber de cuantas maneras se pueden seleccionar m elementos *"mal ubicados"* del conjunto $\mathbb{N}_n$. Aquí no importa el orden de los elementos, por ejemplo, digamos que en la permutación $P_4 = \{3, 1, 2, 4\}$ están *"mal ubicados"* los elementos $(3, 1, 2)$, da igual el orden en que se encuentren dichos elementos mientras cumplan la propiedad de estar *"mal ubicados"*. Como en este caso no nos interesa el

orden de los elementos en la distribución, sino la composición de dicha distribución, estamos en presencia de una combinación de n en m $\binom{n}{m}$. Nótese que solo habrá una manera de tener los elementos restantes *"bien ubicados"*, o sea que conservarán su posición respecto al conjunto $\mathbb{N}_n$. Por eso es innecesario aplicar la regla del producto ya que resultaría en $\binom{n}{m} * 1$.
Ya tenemos de cuantas maneras se pueden seleccionar m elementos *"mal ubicados"* de $\mathbb{N}_n$, necesitamos saber ahora para los índices escogidos cuantas permutaciones p cumplen $p_i \neq i$. El número de permutaciones que cumplen esto puede ser calculado computacionalmente sin problemas ya que $m \leq 4$.
Llamemos d a la función que calcula lo anterior, por lo que dada una $m-tupla$ de elementos cualesquiera, $d(m)$ nos dirá cuantas permutaciones de esta tendrán todos sus elementos *"mal ubicados"*. Entonces para un m fijo si aplicamos la regla del producto obtendremos $\binom{n}{m}d(m)$, que serán la cantidad permutaciones con m elementos *"mal ubicados"*, y como $m \leq k$ estas permutaciones serán casi la identidad. Para obtener la cantidad de permutaciones que son casi la identidad para k debemos aplicar la fórmula obtenida a cada m menor que k, resultando: $\sum_{m=0}^{k} \binom{n}{m} d(m)$.
Demostremos a continuación que no importan los m elementos seleccionados, sino la cardinalidad de la tupla, o sea m.

Para la demostración nos apoyaremos en la función F. Dicha función recibe una $n-tupla$ con sus elementos *"bien ubicados"* y devuelve un conjunto maximal de $n-tuplas$, donde cada una representa una permutación de la original con todos sus elementos *"mal ubicados"*:

$$F(< x_1, ..., x_n >) = \{< q_1 >, ..., < q_j >\}$$

donde cada q_j es una permutación de $< x_1, ..., x_n >$, y todos sus elementos están *"mal ubicados"*.
Provemos que la función F, dadas dos $n-tuplas$ ordenadas de igual cardinalidad, los conjuntos devueltos serán de la misma cardinalidad.

Sea $x =< x_1, ..., x_n >$ y $y =< y_1, ..., y_n >$. Entonces $F(x) = C_1$ y $F(y) = C_2$, donde $|C_1| = |C_2|$. Supongamos que lo anterior no se cumple, digamos sin pérdida de generalidad que $|C_1| < |C_2|$. Veamos como si esto pasa C_1 no será un conjunto maximal.

Sea la función auxiliar A. Esta función biyectiva convierte una $n-tupla$ de tipo y en una de tipo x con los elementos desordenados de la misma forma, o sea, el elemento x_i ocupa la misma posición que el y_i $(1 \leq i \leq n)$, cada uno en su respectiva tupla.

$$A(< y_i, y_j, ..., y_k >) =< x_i, x_j, ..., x_k >$$

donde en una tupla todos los subíndices son diferentes y pertenecen al intervalo $[1, n]$.

Sea la $n-tupla$ $< y_i, y_j, ..., y_k > \in C_2$, si hacemos $A(< y_i, y_j, ..., y_k >)$ da como resultado $x =< x_i, x_j, ..., x_k >$, donde x tiene todos sus elementos *"mal ubicados"*, si este no fuera el caso $\exists x_p \in x$ que está *"bien ubicado"*, entonces por la definición de A $\exists y_p \in < y_i, y_j, ..., y_k >$ que está *"bien ubicado"*, por tanto $< y_i, y_j, ..., y_k > \notin C_2$, llegando a una contradicción. Luego x tiene todos sus elementos *"mal ubicados"*.
$\therefore$ si pasamos la función A por todos los elementos del conjunto C_2, obtendremos todos los elementos de C_1. Pero como $|C_1| < |C_2|$, va a existir al menos un elemento en C_2 que al pasarlo por A dará como resultado una $n-tupla$ de tipo x que no estará en el conjunto C_1. Entonces al conjunto C_1 le falta al menos un elemento, lo que implica que C_1 no es maximal. Llegamos a una contradicción, ya que por definición de F el conjunto C_1 es maximal..$\therefore$ $|C_1| = |C_2|$.

Entonces la función F para cualesquiera $n-tuplas$(de igual cardinalidad) dará como resultado conjuntos de la misma cardinalidad.

Habiendo llegado a este resultado podemos decir que la función d tiene como dominio la cardinalidad de las tuplas que recibe la función F, y como codominio la cardinalidad de los conjuntos devueltos por F. El siguiente ejemplo ilustra lo que estamos diciendo:
Sea la $3-tupla$ $p =< 1, 2, 3 >$

$$F(p) = \{< 2, 3, 1 >, < 3, 1, 2 >\}$$

donde la cardinalidad de p es 3 y la cardinalidad del conjunto devuelto por F es 2. Entonces $d(3) = 2$. Finalmente hemos logrado llegar a que d solo depende de la cardinalidad de los elementos con los que se está trabajando.

1.4. Algoritmos

1.4.1. Fuerza Bruta:

La solución por fuerza bruta consiste en generar todas las permutaciones de tamaño n y comprobar una por una las que cumplen la condición de ser casi la identidad. Evidentemente este algoritmo es ineficiente no solo en tiempo de ejecución, sino también en espacio en memoria, ya que si generamos todas las permutaciones de tamaño n, serían $n!$ arrays de tamaño n. A continuación el algoritmo por fuerza bruta:

```
1   def Count_AIP_BF(n, k):
2       perm = []
3       perm = Permutaciones(n)
4       ans = 0
5       for each p in perm:
6           count = k
7           ans += 1
8           for i in range(n):
9               if p[i] != i+1:
10                  count -= 1
11              if count < 0:
12                  ans -= 1
13                  break
14      return ans
```

La variable *perm* es una lista que guarda todas las permutaciones de tamaño n, por lo que $|perm| = n!$. El algoritmo realiza un recorrido por todos los elementos de *perm* y a su vez recorre cada uno de esos elementos de tamaño n. Las líneas dentro de ambos ciclos se ejecutan en $O(1)$, por lo que la complejidad temporal de este algoritmo será $O(n! * n)$. Este algoritmo no solo es ineficiente en cuanto al tiempo, sino también en cuanto al uso de memoria, ya que tendremos en *perm* $n!$ permutaciones de tamaño n.

1.4.2. Utilizando Combinatoria:

Anteriormente encontramos una fórmula que nos permite dado un m fijo ($0 \leq m \leq k$) y una función d, hallar la cantidad de permutaciones que son casi la identidad.

La función d recibe como parámetro un entero m, y nos devuelve de cuantas formas pueden quedar *"mal ubicados"* los elementos de un array de tamaño m. Trabajemos con los índices de los elementos, que como m está acotado superiormente por 4 serán $(1, 2, 3, 4)$.

Si $m = 0$ esto quiere decir que no pueden haber elementos *"mal ubicados"* en el array original, y solo hay una forma de que esto pase que será el propio array original, por lo que $d(0) = 1$.

Si $m = 1$ esto quiere decir que el array de tamaño m solo tendrá un elemento y no hay forma de que ese elemento pueda estar *"mal ubicado"*, por lo que $d(1) = 0$.

Si $m = 2$ el array de tamaño m tendrá dos elementos y la única forma de que estén *"mal ubicados"* será intercambiándolos, por lo que $d(2) = 1$.

Si $m = 3$ el array de tamaño m tendrá tres elementos con los índices $(1, 2, 3)$, y solo existen dos maneras de que esos índices estén *"mal ubicados"*: $(2, 3, 1)$ y $(3, 1, 2)$, por lo que $d(3) = 2$.

En el caso de $m = 4$ ya es un poco más complicado darse cuenta, por lo que se implementó una función F que lo calcula y obtuvimos el valor que es 9. Más adelante se deja el algoritmo que calcula esto para cualquier m, aunque en nuestro caso solo es hasta 4.

A continuación los algoritmos:

```
def F(m):
    count = 0
    perm = Permutaciones(m)
    for each p in perm:
        count += 1
        for i in range(m):
            if p[i] == i+1:
                count -= 1
                break
    return count
```

```
def Count_AIP_C(n, k):
    count = 0
    for m in range(k+1):
        count += Combinaciones(n,m)*d(m)
    return count
```

$Combinaciones(n, m) = \binom{n}{m}$

Para resolver $\binom{n}{m}$ podemos utilizar el Binomio de Newton, $\frac{n!}{(n-m)!*m!}$. Analicemos esta fórmula:

$$\frac{n!}{(n-m)! * m!} = \frac{n(n-1)(n-2)...(m+1)m!}{(n-m)!m!} = \frac{n(n-1)(n-2)...(m+1)}{(n-m)!}$$

Con la fórmula obtenida pueden ocurrir dos casos, el primero que $m < n-m$ y el segundo que $m \geq n-m$.

- Caso $m < n - m$:

  $$\frac{n(n-1)(n-2)...(m+1)}{(n-m)!} = \frac{n(n-1)(n-2)...(n-m)...(m+1)}{(n-m)...(m+1)m!} = \frac{n(n-1)(n-2)...(n-m+1)}{m!}$$

  Esta operación puede resolverse con un único recorrido de rango m, ya que en el numerador hay m factores al igual que en denominador, por tanto en el mismo recorrido puden realizarse las multiplicaciones. Por tanto la complejidad temporal para este caso es $O(m)$.

- Caso $m \geq n - m$:

  $$\frac{n(n-1)(n-2)...(m+1)}{(n-m)!} \tag{1}$$

  En este caso con un único recorrido de rango $n - m$ puede resolverse la operación, ya que tanto en el numerador como en el denominador hay $n-m$ factores. Esto tiene una complejidad de $O(n-m)$, pero como $m \geq n - m$ podemos decir que es $O(m)$.

Podemos concluir que el costo de realizar $Combinaciones(n, m)$ será $O(m)$. Estamos asumiendo que la multiplicación es constante, que en este caso lo es ya que tanto n como m están acotadas, y como $m \leq k$, digamos que es $O(k)$.
Entonces como el ciclo for recorre un rango de $k + 1$, el costo de calcular las combinaciones es $O(k)$ y $d(m)$ se resuelve en $O(1)$, el costo total del algoritmo será $O(k^2)$, y $k \leq 4$ por lo que el algoritmo se resolverá en un tiempo constante.

1.4.3. Tester:

Se ha implementado un Tester para comprobar la correctitud del segundo algoritmo implementado ($Count\_AIP\_C$). El tester genera valores random de n y k y valida la respuesta en relación a la obtenida del algoritmo por fuerza bruta, luego se imprime la cantidad de casos correctos que hubo, y en caso de resultar alguno incorrecto se imprimirá dicho caso. A continuación la implementación:

```
def Tester(tests_number, ran_n):
    corrects = 0
    incorrects = []
    for i in range(tests_number):
        n = random(4, ran_n)
        k = random(1, 4)
        if (Count_AIP_C(n,k) == Count_AIP_BF(n,k))
            corrects += 1
        else:
            incorrects.Insert((n,k))
    print(corrects + "CasosCorrectos")
    print("CasosIncorrectos" + incorrects)
```

La variable *tests_number* indica la cantidad de casos de prueba que queremos generar, *ran_n* será la cota superior para n. Como el Tester se apoya en el algoritmo por fuerza bruta cabe señalar que para valores elevados de *ran_n* es muy probable que de error en memoria, ya que el algoritmo por fuerza bruta genera $n!$ permutaciones de tamaño n.

2. Bibliografía:

- N. Vilenkin. Arreglos, Permutaciones y Combinaciones. ¿De Cuantas Formas?