

➤ Ejemplo de solución del problema.

Se tomarán números del 1 al 3 para facilitar la comprensión, pero todo el código esta como la orden lo plantea, del 1 al 8.

Sea la lista de entrada:

1	1	2	3	2	3	1	3
---	---	---	---	---	---	---	---

Tenemos tres condiciones que cumplir:

- Si ya se tiene un numero en la lista de resultado, todos sus iguales tienen que venir consecutivos, y sus respectivos índices en la lista original tienen que estar en orden creciente

1	2	3
---	---	---

1	1	2	3	3	3
---	---	---	---	---	---

1	1	2	2	3	3
---	---	---	---	---	---

- La subsecuencia resultante tiene que tener la mayor cardinalidad posible:

1	1	2	3	3	3
---	---	---	---	---	---

1	1	2	2	3	3
---	---	---	---	---	---

- Si uno de los números del 1 al 8 no aparece en la solución, esto cuenta como cardinalidad 0 en el resultado.
- Para cada subconjunto en la lista de resultado la diferencia entre sus cardinalidades tiene que ser a lo sumo 1:

1	1	2	2	3	3
---	---	---	---	---	---

Si cumplen estas 3 condiciones el problema está resuelto, en el caso de este ejemplo la solución es la última lista mostrada.

➤ Ideas abordadas:

- Combinatoria:

La primera idea que se pensó fue resolver el problema por combinatoria, se generan todas las posibles combinaciones que puede tener la lista de entrada y luego se verifica cual/es cumplen las condiciones antes mencionadas. Este algoritmo fue implementado como tester y será explicado en detalle más adelante.

- Subsecuencia máxima creciente:

Luego se el problema fue modelado de forma tal que se resolviera usando el algoritmo de máxima subsecuencia creciente. La idea fue:

Tomar la lista de entrada del problema (A partir de ahora esta lista se denotará como L)

1	1	3	2	3	2	1	2
---	---	---	---	---	---	---	---

Junto con sus índices

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

Luego se agrupan los elementos de la lista en orden de aparición y se guardan en una lista la actualización de los índices que dio como resultado construir esta permutación

Permutación (A partir de ahora esta lista se denotará como P)

1	1	1	3	3	2	2	2
---	---	---	---	---	---	---	---

Índices actualizados (A partir de ahora esta lista se denotará como I)

0	1	6	2	4	3	5	7
---	---	---	---	---	---	---	---

Al obtener esta lista de "índices" se garantiza que $\forall x, y \in I$ si $x < y$ significa que x aparece antes que y en L .

Por tanto si se obtiene la máxima subsecuencia creciente de I

0	1	2	3	5	7
---	---	---	---	---	---

se garantiza que si $x > y$, x no va a aparecer antes que y en la solución, la cual es una de las condiciones del ejercicio, además, como I se obtuvo a partir de los cambios que llevaron de L a P y en este cada subconjunto de números está garantizado que $\forall x, y \in P : x = y$, x es consecutivo con y , en I ocurre lo mismo y por ende una solución usando este algoritmo también cumpliría esta propiedad. Hasta este punto el problema podría ser resuelto en $O(n \log n)$, pero no se encontró manera de mantener un tiempo similar a este añadiendo el cumplimiento de la restricción de la diferencia de las cardinalidades ya que esto implicaba tener que usar backtrack, por lo que esta solución fue descartada.

- Optimo (BS + Greedy + DP):

Idea general de la solución:

- 1- Se toman todos los posibles ordenes en los que pueden aparecer cada conjunto en la solución (todas las permutaciones de $[1 \dots 8]$).
- 2- Para cada una se busca cual es el tamaño de la subsecuencia máxima que se pudo encontrar

- 3- De los máximos de tamaño de cada una de las permutaciones, la solución es el mayor entre ellos.

Demostracion:


- Sea A la lista de entrada tal que $\forall i \in A \ 1 \leq i \leq 8$
- Sea S la subsecuencia máxima que cumple las restricciones (La solución óptima del problema).
- Si se toma de S la primera aparición de cada número diferente, dicha subsecuencia va a pertenecer a una de las permutaciones.
- Como S es máxima no puede existir otra solución opima con una permutación diferente que tenga un mayor tamaño porque si no S no fuera máximo
- Por tanto, dada una permutación, si de esta se obtiene la subcadena de mayor tamaño, esa es la solución del problema.

El algoritmo se verá en el apartado de ALGORITMOS.

➤ Generador + Tester:

Se crearon 2 archivos .txt donde uno contiene los casos de prueba y el otro las soluciones de estos dadas por el algoritmo combinatorio que se verá más adelante.

- Generador:



```
1 def Generator():
2     res = []
3     for i in range(100):
4         tam = rd.randint(0,20)
5         list = []
6         for j in range(tam):
7             num = rd.randint(1, 8)
8             list.append(num)
9         res.append(list)
10    return res
```

Este algoritmo genera todo tipo de casos de prueba validos ya que las listas de entrada son de tamaño arbitrario, en este caso se toma un numero aleatorio entre 0 y 20 debido a que el algoritmo combinatorio demora mucho en resolver el problema para tamaños mayores. La otra condición es que se genere un número aleatorio del 1 al 8 para cada posición de la lista, y estos pueden repetirse cualquier cantidad de veces y en cualquier orden, el método lo resuelve generando números aleatorios para cada posición de la lista.

- Tester:

```
1 def Tester():
2     test, sol = LoadTxt()
3     for i in range(len(test)):
4         res = optimal(list(test[i][0:len(test[i])-1]))
5         if(res == int(sol[i])):
6             print(f'Case {i}: OK')
7         else:
8             print(f'Case {i}: Wrong')
```

Se guardan en una lista los casos prueba con sus respectivas soluciones y luego al algoritmo optimo se le pasan cada uno de los casos y se verifica si su resultado es correcto.

➤ Algoritmos:

- Combinatoria (Fuerza bruta):

El primer algoritmo pensado fue el más intuitivo, el cual es una solución muy ineficiente, pero es la más sencilla de demostrar su correctitud y complejidad. Esta es la solución por combinatoria, la cual consiste en generar todos los posibles resultados y devolver el/los que cumplan los requisitos anteriormente mencionados.

```
1 from itertools import combinations
2
3 def Combinations(arr):
4     comb = []
5     for i in range(len(arr)):
6         comb.append(combinations(arr, i+1))
7     return comb
```

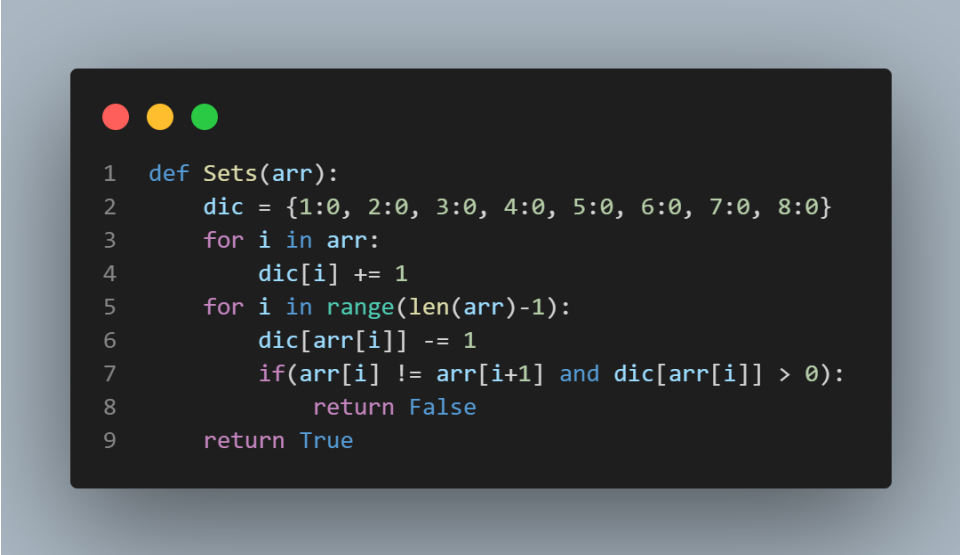
En este fragmento se utiliza el módulo `itertools.combinations`, el cual hace uso del método `combinations`, este recibe una lista y un valor k (tamaño de la combinación). Como el objetivo es generar todas las posibles combinaciones de la lista de entrada basta con generar todas las combinaciones variando el número de k entre $1 \leq k \leq n$ donde n es el tamaño de la lista, esto se realiza entre las líneas 5 y 6. Las combinaciones generadas se guardan en una variable y se devuelven para luego analizar el cumplimiento de las restricciones.

El costo de buscar todas las combinaciones para un valor de k es $2^n - 1$ y como se hace esto para cada valor de k hasta n , la complejidad de este método es:

$$O(n * 2^n)$$

Luego de obtener todas las combinaciones, se verifica cual o cuales cumplen las condiciones:

Condición 1:



```
1 def Sets(arr):
2     dic = {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0}
3     for i in arr:
4         dic[i] += 1
5     for i in range(len(arr)-1):
6         dic[arr[i]] -= 1
7         if(arr[i] != arr[i+1] and dic[arr[i]] > 0):
8             return False
9     return True
```

Este método primeramente cuenta cuantas veces se repite cada elemento de la lista y lo guarda en un diccionario, eso es $O(n)$. Luego como se quiere saber si los números que son iguales en la solución aparecen de forma consecutiva, lo que se hace es recorrer la lista nuevamente y se verifica si la cantidad de elementos consecutivos coincide con el valor guardado en el diccionario, en caso de que el algoritmo recorra la lista completa significa que no encontró problemas y la solución es válida, en caso contrario si la cantidad de elementos consecutivos es menor que la cantidad de ocurrencias totales significa que existe al menos un conjunto de números entre los consecutivos verificados y los que faltaron, por lo que esto es invalido. La complejidad temporal de este método es:

$$O(n + n) = O(n)$$

Condición 3:

```
1 def NotAll(arr):
2     count = 0
3     dic = {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0}
4     for i in arr:
5         dic[i] += 1
6     for i in dic:
7         if(dic[i] > 0):
8             count += 1
9     if(count == 8):
10         return -1
11     return count
```

Este método verifica si cada número del 1 al 8 aparece al menos una vez en la lista de entrada, ya que en caso de no ser así, en la lista de resultado solo podrá existir a lo sumo 1 de cada número que este en la entrada y el resultado sería $8 - x$ donde x es la cantidad de números del 1 al 8 que no aparecen. Este método es $O(n + 8) = O(n)$

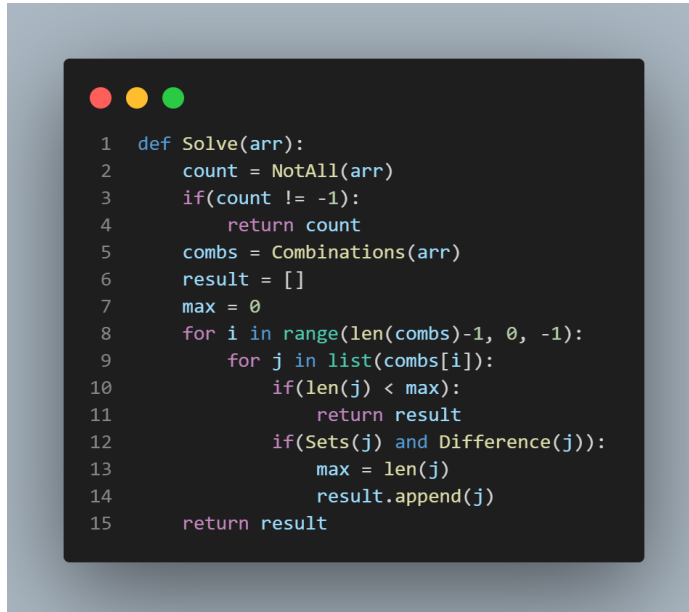
Condición 4:

```
1 def Difference(arr):
2     dic = {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0}
3     for i in arr:
4         dic[i] += 1
5
6     dif = [dic[i] for i in dic]
7
8     if(max(dif) - min(dif) > 1):
9         return False
10    return True
```

Este método, al igual que el anterior cuenta las ocurrencias de cada elemento de la lista y como se quiere verificar si las cardinalidades para todo par de conjunto de elementos difieren en a lo sumo 1, basta con obtener cual es el conjunto que tiene más elementos y cuál es el que menos, si la diferencia es mayor 1 no se

cumple la restricción por tanto no es solución, en caso contrario si pertenece. Hacer esta resta funciona debido a que todas las cardinalidades de los conjuntos están acotadas por: $\min \leq x_i \leq \max : 1 \leq i \leq 8$ donde x_i es la cardinalidad asociada al conjunto i , así que si la diferencia entre el mínimo y el máximo es menor o igual que 1, también lo será para cualquier elemento entre el máximo y otro mayor que el mínimo y viceversa. La complejidad temporal de este método es $O(n + 8 + n) = O(n)$

Condición 2:

A screenshot of a code editor with a dark background and light-colored text. The code is a Python function named 'Solve' that takes an array 'arr' as input. It first calls 'NotAll(arr)' and checks if the result is not -1. If so, it returns the count. Otherwise, it generates combinations of the array, iterates through them from largest to smallest, and checks if they meet the conditions. The code is as follows:

```
1 def Solve(arr):
2     count = NotAll(arr)
3     if(count != -1):
4         return count
5     combs = Combinations(arr)
6     result = []
7     max = 0
8     for i in range(len(combs)-1, 0, -1):
9         for j in list(combs[i]):
10            if(len(j) < max):
11                return result
12            if(Sets(j) and Difference(j)):
13                max = len(j)
14                result.append(j)
15     return result
```

En el caso mejor: Si no aparece al menos uno de cada número en la entrada, ya que de ser así el resultado es el que se explicó anteriormente sin necesidad de buscar combinaciones u otra verificación es $O(n)$

Esta condición se resuelve en la ejecución del método principal del algoritmo ya que, si el objetivo es encontrar la subcadena máxima, basta con recorrer la lista de todas las posibles combinaciones, la cual esta ordenada de forma creciente (ver Fig1) comenzando desde la última posición hasta el comienzo, de esta forma la primera cadena que cumpla las otras dos condiciones explicadas anteriormente será la de tamaño máximo. (En este caso, esta solución fue la implementada para el tester, por tanto, se le añadió que encontrara todas las cadenas máximas ya que esta puede no ser única y no tener problemas con algoritmos más eficientes que solo dan una óptima). La complejidad del algoritmo en general es la complejidad de este método en un caso que no sea el mejor (el peor) es: $O(2^n * (n + n)) = O(n * 2^n)$

- Optimo:

Este es el algoritmo más eficiente pensado, si idea general fue la explicada anteriormente, pero ahora será vista en más detalle.

```

1  def NotAll(arr):
2      count = 0
3      dic = {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0}
4      for i in arr:
5          dic[i] += 1
6      for i in dic:
7          if(dic[i] > 0):
8              count += 1
9      if(count == 8):
10         return -1
11     return count

```

Si en la lista de entrada no existe al menos una ocurrencia de cada elemento, al igual que en la solución por combinatoria el resultado es tomar un elemento por cada uno que aparece, y este es el mejor caso y se resuelve en $O(n)$.

```

1  def Minimum(arr): #O(n)
2      dic = {'1':0, '2':0, '3':0, '4':0, '5':0, '6':0, '7':0, '8':0}
3      for i in arr:
4          dic[i] += 1
5
6      dif = [dic[i] for i in dic]
7
8      return min(dif)

```

Sea x la cantidad de veces que se repite el número que menos se repite de la lista de entrada, en el óptimo no puede existir ningún conjunto con cardinalidad mayor a $x + 1$. Esto sirve como cota para mejorar el algoritmo que sea explicado más adelante.

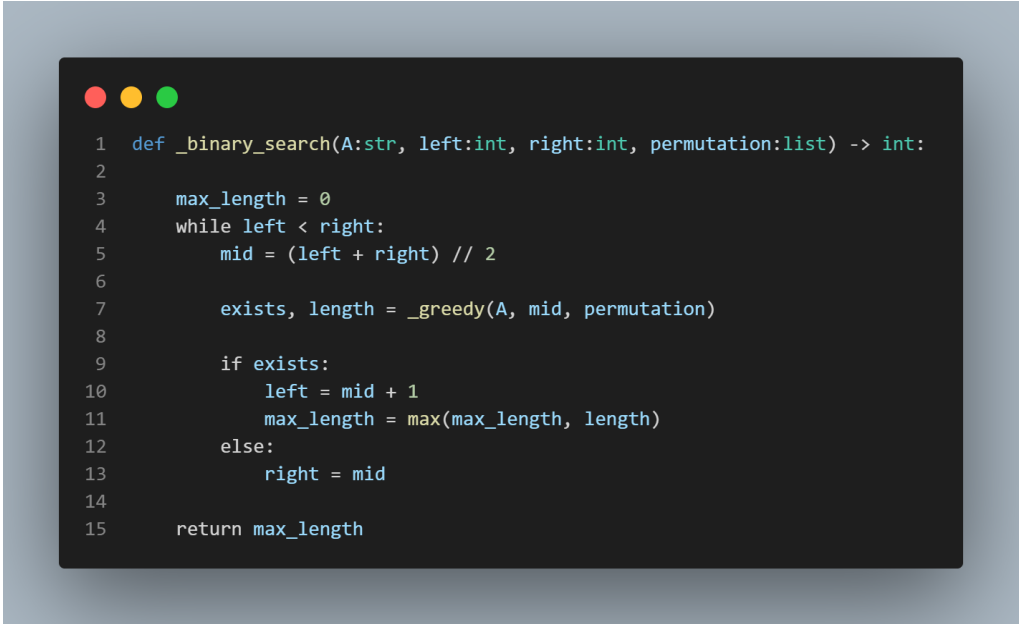
Demostración:

Como la diferencia de las cardinalidades de los conjuntos en el resultado tiene que ser a lo sumo uno, no existe manera de que si el mínimo es x en la solución exista un conjunto con cardinalidad $x + i : i \geq 2$ ya que:

- si el número que se repite x veces se pone esa cantidad de veces en la solución, los restantes solo pueden repetirse $x + 1$ o $x - 1$ veces
- si el número que se repite x veces se pone una cantidad menor de veces en la solución, los restantes solo pueden repetirse $x - k + 1$ o $x - k - 1$ veces donde $x - k$ es el numero menor que x que se repitió.
- si el número no forma parte de la solución, al contar como cardinalidad 0 ocurre exactamente lo mismo.

Complejidad: $O(n)$.

Ahora se verá la utilidad de tener este valor:



```
1 def _binary_search(A:str, left:int, right:int, permutation:list) -> int:
2
3     max_length = 0
4     while left < right:
5         mid = (left + right) // 2
6
7         exists, length = _greedy(A, mid, permutation)
8
9         if exists:
10             left = mid + 1
11             max_length = max(max_length, length)
12         else:
13             right = mid
14
15     return max_length
```

- (1) En el momento que se tenga fijada una permutación, hay que probar si se puede construir la subsecuencia máxima tal que todos los conjuntos de la solución tengan como cardinalidad mínima a x y máxima a $x + 1$ (Esta corresponde a la misma x que en el ejemplo anterior, y más adelante se explica cómo lograr esto), en caso de no existir solución válida para estos valores, hay que disminuir el valor de x en 1 y volver a probar, esta sería una búsqueda común en un ciclo disminuyendo en 1 por cada iteración mientras la condición no se cumpla. Esta solución es $O(n * k)$ donde k es el costo de obtener la máxima subsecuencia que se está buscando.
- (2) Una solución más interesante sería aplicar búsqueda binaria sobre este valor de x , ya que es la forma mas eficiente y es perfectamente adaptable a esta situación.

Demostración:

En (1) se realiza una búsqueda donde el primer valor es x y en cada iteración este valor disminuye en 1.

Sea $k : k < x$ el primer valor donde se cumple la condición, este es el mejor para la permutación fijada ya que en los valores entre k y x no cumplían, y cualquier valor menor que k empeoraría la solución.

Esto es lo mismo que buscar cual el primer valor que cumple una propiedad en una lista ordenada, de aquí que se pueda aplicar búsqueda binaria para mejorar este proceso. La búsqueda consistiría en:

Sea m el valor tomado como mid en la búsqueda binaria, si no podemos construir ninguna subsecuencia valida tomando como mínimo el valor de m , tampoco se podrá con un valor mayor, por lo que la próxima iteración tendrá un valor de m menor. En caso de haber encontrado una subsecuencia tomando m como mínimo,

entonces hay que ver si es posible encontrar una para un mayor valor de m , por lo que en la próxima iteración, su valor será mayor siguiendo el algoritmo de búsqueda binaria clásica.

La complejidad de esto es $O(\log n)$