Parcial2-ADA

Daniel F Osorio

Mayo 3 del 2021

1 Punto 1, apartado a:

1.1 Descripción y correctitud del algoritmo:

Entrada: Arreglos $L[0..N]$ y $R[0..N]$ de números reales con $N \geq 0$, donde para cada $0 \leq n < N$, $(L[n], R[n])$ representa el intervalo n que pertenece al conjunto de intervalos X.

Salida: Conjunto mínimo de puntos P que atraviesen todos los intervalos del conjunto X

Inicialmente, para este algoritmo se hará el siguiente cambio en la entrada:

- La representación del conjunto de intervalos X por medio de los arreglos L y R, se hará ahora por medio de un arreglo de tuplas A tal que $\forall_n(0 \leq n < N \Rightarrow A[n][0] = L[n] \wedge A[n][1] = R[n])$

De esta manera se tiene que:

Nueva Entrada: Arreglo $A[0..N]$ de tuplas (s_i, e_i) con s_i, e_i números reales y con $N \geq 0$, donde para cada $0 \leq n < N$, $(A[n][0], A[n][1])$ representa el intervalo n que pertenece al conjunto de intervalos X.

Además de esto, se tendrá la siguiente convención:

- el elemento (s, e) de X representa el intervalo cerrado $[s, e]$. De esta forma, dos intervalos (s_1, e_1) y (s_2, e_2) se intersectan si $(s_1, e_1) \cap (s_2, e_2) \neq \emptyset$

Ahora bien, para resolver este problema se utilizara un algoritmo voraz. La idea es conseguir un conjunto con la menor cantidad de puntos que atraviesen todos los intervalos. Sin embargo, la combinación de puntos que cumplen esta condición no es única y pueden ser infinitas combinaciones. Si se toma de ejemplo $X = \{(s, e)\}$, la respuesta para esta entrada podría ser todo punto p, tal que $s \leq p \leq e$, y como son números reales podrían ser infinitos puntos. Sin embargo, lo que si es único es el intervalo donde se encuentran estos puntos, en este caso (s, e). De esta forma, la idea es buscar el conjunto con la menor

cantidad de intervalos, de los cuales si se selecciona un punto por cada intervalo, se puede atravesar todos los intervalos de X. En otras palabras, lo que se busca es un conjunto mínimo de intervalos I tal que:

- I contiene la menor cantidad de intervalos posibles que atraviesan a X.
- Para todo intervalo $(s, e) \in X$, existe un intervalo $(s_i, e_i) \in I$ tal que contiene un punto p que también esta incluido en el intervalo (s, e), es decir, $(s, e) \cap (s_i, e_i) \neq \emptyset$.
- Para todo intervalo $(s_i, e_i) \in I$ se cumple que todo punto p tal que $p \in (s_i, e_i)$, entonces p debe pertenecer también a algún intervalo $(s, e) \in X$.

De esta forma, para construir el algoritmo se propone el siguiente teorema de optimización local:

Teorema de optimización local: Sea X un conjunto finito de intervalos, sea n la cantidad mínima de intervalos que se requieren para atravesar X y sean $x_1, x_2...x_n \subseteq X$ tales que representan los grupos de intervalos en los cuales se divide X al atravesarlo. Esto se puede visualizar mejor con la siguiente imagen:

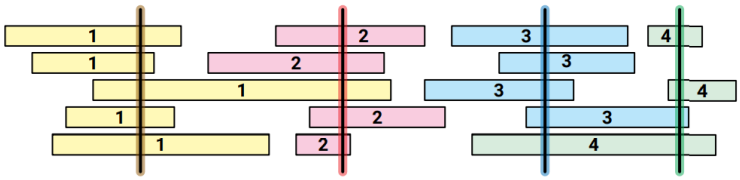

Figure 1: en este caso $n = 4$ y x_1, x_2, x_3, x_4 se representan por los números 1,2,3 y 4 respectivamente

Continuando entonces con el teorema, sea el intervalo (s, e), en el cual s es el punto máximo de iniciación de algún subconjunto x_i y e el punto mínimo de terminación del subconjunto x_i, entonces (s, e) pertenece al conjunto de intervalos mínimo I, que atraviesa a X. Si se toma en cuanta la imagen anterior, esto se vería de la siguiente forma:

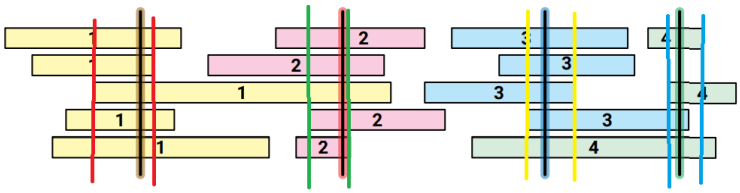

Figure 2: Las lineas de distintos colores marcan los limites de los intervalos del conjunto I

Demostración: Sea $(s, e) \in I$ tal que (s, e) atraviesa a $x_i \subseteq X$. Supongamos por contradicción que s no es el máximo punto de inicio de los intervalos en x_i o que e no es el mínimo punto de terminación de los intervalos en x_i. Si s no es máximo, entonces existe un punto s' en un intervalo de x_i (sea este (s', e')) tal que $s' > s$, por lo cual, si $e < s'$, entonces $s' \notin (s, e)$ y $(s, e) \cap (s', e') = \emptyset$, lo cual contradice el hecho que (s, e) atraviesa a x_i. Por otro lado, si e no es mínimo, entonces existe un punto e' en un intervalo de x_i (sea este (s', e')) tal que $e' < e$, por lo cual, si $s > e'$, entonces $e' \notin (s, e)$ y $(s, e) \cap (s', e') = \emptyset$, lo cual contradice el hecho que (s, e) atraviesa a x_i. $\square$

De esta forma, utilizando continuamente este teorema de optimización local se busca conseguir el conjunto I con los intervalos, para después escoger cualquier punto de cada uno de los intervalos y retornarlo como respuesta. En el caso de este algoritmo, lo que se hará es escoger siempre el limite superior de los intervalos de I, de esta forma si $I = \{(s_1, e_1), (s_2, e_2)...(s_n, e_n)\}$ entonces el conjunto de puntos de salida $P = \{e_1, e_2...e_n\}$. Se propone el siguiente pseudocódigo:

```
1) ordenar los intervalos de A de menor a mayor segun
   su limite superior.
2) si A = ∅ entonces la respuesta es ∅
3) De lo contrario:
   SOLVE(A, e, n):
      1) Si n = N-1 retornar {e}
      2) De lo contrario:
        a) Si el siguiente intervalo A[n+1] = (s',e')
           tiene tiempo de inicio menor o igual
           al tiempo de finalizacion actual e (s' < e),
           se retorna SOLVE(A, min(A[n+1][1], e), n+1)
        b) De lo contrario,  se retorna
           {e} ∪ SOLVE(A, A[n+1][1], n+1)
```

En este algoritmo, los casos base cuando $X = \emptyset$ y $X = \{(s, e)\}$ se tuvieron que separar, ya que SOLVE no podría ejecutarse si $X = \emptyset$, ya que no habría un limite superior e para hacer el llamado. La idea es recorrer los intervalos de X en orden e ir identificando para los subconjuntos $x_i \in X$, el valor e tal que e sea el limite superior de su intervalo correspondiente en I. De esta forma se puede cambiar la salida del algoritmo:

Nueva salida: Ordenar los intervalos de A de menor a mayor según su limite superior. Si $A = \emptyset$ entonces $\emptyset$, de lo contrario SOLVE(A, A[0][1], 0).

Ahora bien para probar la correctitud del algoritmo se probara el siguiente teorema de optimización global:

Teorema de optimización global: El algoritmo construye un conjunto de puntos P mínimo que atraviesa X, donde todo $p \in P$ es el punto final de un intervalo respectivo del conjunto de intervalos I que atraviesa a X.

Demostración: Se hará una demostración por inducción:

Casos bases:

- $\boldsymbol{X = \emptyset}$**:** Si X es vació entonces $P = \emptyset$ ya que no hay ningún intervalo,lo cual se cumple con la salida del algoritmo.
- $\boldsymbol{X = (s, e)}$**:** Si X tiene un solo elemento, entonces $I = \{(s, e)\}$ y $P = \{e\}$, lo cual se cumple con la salida del algoritmo.

Caso inductivo:

Sea $X = \{(s_1, e_1)...(s_N, e_N)\}$ el conjunto de intervalos ya organizado, es decir que $\forall_{i,j}(i < j \Rightarrow e_i \leq e_j)$. Ahora bien sea $I = \{(s'_1, e'_1)...(s'_m, e'_m)\}$ el conjunto mínimo de intervalos para atravesar X. Por ver entonces que el algoritmo retorna $P = \{e'_1...e'_m\}$. Inicialmente entra el primer intervalo del conjunto de intervalos ordenado X, representado por $A[n]$ con terminación en e_n. Si el intervalo siguiente $A[n+1]$ tiene tiempo de inicio $A[n+1][0] \leq e_n$, entonces $A[n] \cap A[n+1] \neq \emptyset$ (ya que A esta ordenado ascendentemente según los limites superiores), por lo cual ambos intervalos deben pertenecer al mismo subconjunto x_i que se genera al atravesar X con I. A continuación se demuestra esta afirmación:

Lema 1: Si $A[n] \cap A[n+1] \neq \emptyset$ entonces ambos intervalos deben pertenecer al mismo subconjunto x_i que se genera al atravesar X con I.

Demostración: Supongamos por contradicción que $A[n]$ y $A[n+1]$ no pertenecen al mismo subconjunto x_i. De esta forma, existirían dos subconjuntos x_k y x_j tales que contengan a $A[n]$ y $A[n+1]$ respectivamente. Sin embargo, esto contradice el hecho que I es mínimo, ya que como $A[n] \cap A[n+1] \neq \emptyset$, entonces se podría reducir en uno la cantidad de intervalos que contiene I, al unir $A[n]$ y $A[n+1]$ en un mismo subconjunto de X. $\square$

De esta manera, el algoritmo une a $A[n+1]$ en el mismo subconjunto x_i en el cual esta $A[n]$, y se calcula el nuevo limite superior del intervalo que lo contiene, que es el mínimo entre el limite superior que se tenia antes para x_i (e_n) y el limite superior de $A[n+1]$ que es $A[n+1][1]$. Luego por teorema de optimización local, podemos asegurar que para x_i, el limite superior del intervalo que lo atraviesa $e'_i = min(ant, A[n+1][1])$, donde anterior era el mínimo limite superior que se tenia antes para x_i. Luego con la recursion se continua con el resto del algoritmo. **(observación 1)**.

En caso de que $A[n+1][0] > e_n$, entonces $A[n] \cap A[n+1] = \emptyset$ (ya que A esta ordenado ascendentemente según los limites superiores), por lo cual ambos intervalos no pueden pertenecer al mismo subconjunto x_i que se genera al atravesar X con I. A continuación se demuestra esta afirmación:

Lema 2: Si $A[n] \cap A[n+1] = \emptyset$ entonces ambos intervalos no pueden pertenecer al mismo subconjunto x_i que se genera al atravesar X con I.

Demostración: Supongamos por contradicción que $A[n] = (s_1, e_1)$ y $A[n+1] = (s_2, e_2)$ pertenecen al mismo subconjunto x_i. De esta forma existe un intervalo $(s, e) \in I$ tal que atraviesa a $A[n]$ y a $A[n+1]$. Sin embargo como $A[n] \cap A[n+1] = \emptyset$ y $e_1 < e_2$ (por que están en orden), entonces $(s, e) \cap (e_1, s_2) \neq \emptyset$, lo que quiere decir que existen puntos $p \in (s, e)$, tal que $p \notin (s_1, e_1)$ y $p \notin (s_2, e_2)$, lo cual contradice el hecho de que todos los puntos de (s, e) deben estar incluidos en los intervalos que contiene el subconjunto x_i. $\square$

Luego, como no pertenecen al mismo x_i y A esta organizado, entonces no hay mas intervalos que puedan estar en x_i, por lo cual se une a P el limite superior que se lleva calculado hasta el momento e_n y se continua con la recursion. **(observación 2)**

Finalmente por observación 1, 2 y por hipótesis de inducción, es posible concluir que si los subconjuntos que se forman al atravesar X con I son $x_1, x_2 ... x_m$, luego el algoritmo calcula el limite superior para cada x_i y lo añade a P, dando así el conjunto $P = \{e'_1 ... e'_m\}$. $\square$

1.2 Calculo complejidad:

El algoritmo central tiene una complejidad de $O(n)$ ya que recorre el arreglo A una vez, mientras que la operación de ordenamiento del principio toma un tiempo de $O(nlog(n))$, por lo cual la complejidad final del algoritmo es de $O(nlog(n))$.

2 Punto 1, apartado b:

Para el algoritmo del apartado a, se plantea la siguiente implementación en Python:

```
def solve(A, N, n, t):
  ans = None
  if(n == N-1): ans = {t}
  else:
    if(A[n+1][0] <= t): ans = solve(A, N, n+1, min(A[n+1][1], t))
    else: ans =  {t}.union(solve(A, N, n+1, A[n+1][1]))
  return ans

def intStab(A):
  A.sort(key = lambda x: x[1])
  N = len(A)
```

```
12    ans = None
13    if(N == 0): ans = {}
14    else: ans = solve(A, N, 0, A[0][1])
15    return ans
```

Esta implementación del algoritmo refleja claramente todos los casos del algoritmo planteado anteriormente:

1. Inicialmente se ordena el arreglo A según el tiempo de finalización de los intervalos con la funcion sort.
2. Después se entra al caso base que es si $A = \emptyset$, lo cual se hace al preguntar si el tamaño de A es cero. Si es cero, entonces debe retonar $\emptyset$, representado por el conjunto vació en Python.
3. En caso que $A \neq \emptyset$, entonces se llama a SOLVE
4. Si A solo tiene un elemento, es decir si $n == N - 1$, entonces se retorna el intervalo superior t que se lleva hasta el momento.
5. En caso que A tenga mas elementos, entonces se hace la comparación si el siguiente elemento pertenece al mismo x_i, con la comparación A[n+1][0] $\leq$ t, ya que si se cumple entonces los intervalos se intersectan. De este forma se une el intervalo n a x_i y se calcula el nuevo limite superior de x_i al llamar solve(A, N, n+1, min(A[n+1][1], t)).
6. En caso de que no se intersecten (A[n+1][0] $>$ t), entonces se concluye que t es el limite superior de un intervalo de I y se continua con el resto de intervalos, por lo cual la implementación retorna {t}.union(solve(A, N, n+1, A[n+1][1])).

De esta forma, intStab(A), resolvería el problema planteado.

3 Punto 2, apartado a:

3.1 Descripción del algoritmo:

Entrada: Un entero positivo n.

Salida: la cadena de sumas que sume n y que sea de tamaño mínimo.

Para este algoritmo se utilizara la estrategia de bactracking, la idea es generar las posibles combinaciones de cadenas de sumas que sumen n y seleccionar la que tenga menor longitud. La idea es iniciar la cadena de suma desde el numero 1. Luego a partir de esta se generan los posibles números que se pueden obtener de la cadena de suma anterior, en este caso seria solamente 2, lo cual daría la cadena [1,2]. luego se hace lo mismo con la cadena [1,2], lo cual generaría los números 3,4 y las cadenas [1,2,4], [1,2,3]. La idea es repetir este proceso hasta que los siguientes números calculados sean n o sean mayor que n (Esta sera nuestra poda, la cual reducirá el árbol de soluciones). Esto se puede observar mejor en los siguientes gráficos:

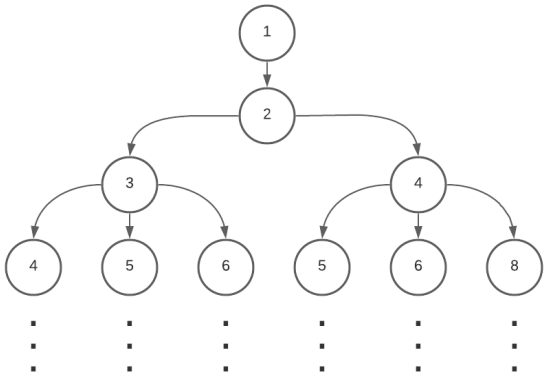

Figure 3: Este árbol muestra como se expanden las posibles soluciones en el algoritmo

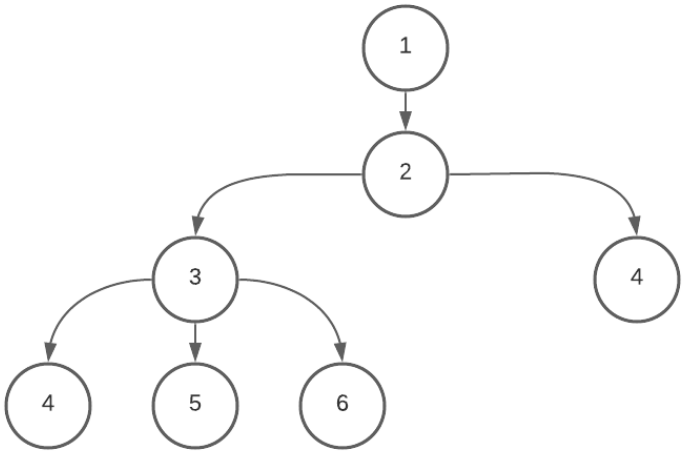

Figure 4: Este seria el árbol de soluciones ya podado para $n = 4$. En este caso, se tiene dos hojas con el 4, por lo cual los caminos que conducen a estas hojas son posibles soluciones ([1,2,3,4] y [1,2,4]). Finalmente se escoge la de menor longitud, por lo cual la respuesta seria [1,2,4].

De esta manera se propone el siguiente pseudocódigo que utiliza esta idea para resolver el problema:

```
1) MIN = ∞
2) ANS = None
addChain(n, k, sol):
  1)Si k = 0 entonces add 1 a sol y addChain(n, 1, sol)
  2)De lo contrario si n = k
    a) Si MIN > longitud de sol entonces ANS = sol y
       MIN = longitud sol
  3)De lo contrario:
    a)por cada valor i en sol:
      +) Si k + i <= n entonces
        -) add k + i a sol
        -) addChain(n, k + 1, sol)
        -) quitar k + i de sol
```

3.2 Explicación del pseudocódigo:

Inicialmente la variable MIN contendrá la longitud mínima de la cadena de sumas que se tiene hasta el momento. En este caso, se esta guardando como la longitud total de la cadena solución, es decir, sin restar el 1. La variable ANS contendrá la cadena de sumas mínima que se tiene hasta el momento. Inicialmente como no hay solución, MIN se inicia en ∞ y ANS en None. En cuanto a los parámetros, n es el mismo de la entrada, k simbolizara el nodo donde me encuentro en el árbol de soluciones y sol el camino que he recorrido para llegar al nodo k. A continuación se describe cada caso:

- Cuando k = 0, quiere decir que apenas se esta empezando, por lo cual se añade 1 a la sol y $k = 1$ para después seguir sumando con addChain.
- Cuando $n = k$, quiere decir que el nodo donde estoy es una posible solución, por lo cual se compara con el mínimo que se lleva hasta ese instante con la longitud del camino para llegar a k, es decir, la longitud de sol. De esta forma si el camino es menor, entonces k es mínimo y se toma como solución por el momento (ANS = sol y MIN = longitud sol).
- Finalmente, si me encuentro en un k intermedio lo que hago es generar los los hijos de este nodo, al generar los siguientes caminos posibles al sumar todos los valores de sol con k. Luego utilizando la estrategia de backtracking, recorro los hijos en busca de la solución con addChain y después me devuelvo en el árbol para generar los demás hijos de otros k intermedios.

De esta forma se puede cambiar la salida del algoritmo:

Nueva salida: la cadena de sumas mínima y longitud mínima de esa cadena para n son ANS y MIN - 1 respectivamente, despues de ejecutar addChain(n, 0, [ ]).

3.3 Implementación en Python:

A continuación se presenta una implementación en Python del algoritmo descrito:

```
MIN = float('inf')
ANS = None
def addChain(n, k, sol):
  global MIN, ANS
  if(k == 0):
    sol.append(1)
    addChain(n, 1, sol)
  elif(n == k):
    if(MIN > len(sol)):
      ANS = sol.copy()
      MIN = len(sol)
  else:
    for i in sol:
      if(k + i <= n):
        sol.append(k + i)
        addChain(n, k + i, sol)
        sol.pop()
```

4 Punto 2, apartado b:

4.1 Descripción del algoritmo:

Entrada: Un entero positivo n y un arreglo de números $A[0..N)$ con $N \geq 0$.

Salida: Enumerar todas las cadenas de sumas de tamaño n que hay en A.

Este algoritmo se basa en una modificación del algoritmo del punto anterior. La idea es expandir el árbol de soluciones como se mostró en la figura 3 y de este escoger las soluciones que tengan tamaño n y que tengan todos sus elementos en el arreglo A. De esta manera, la poda consiste en expandir el árbol solo a los nodos que pertenezcan a A y llegar hasta el nivel $n+1$ del árbol (debe ser $n+1$ ya que la longitud de la cadena de sumas no cuenta el 1). Por ejemplo si se tiene el arreglo $A = [1, 2, 3, 4, 5]$ y $n = 3$, entonces las soluciones a listar serian [1, 2, 3, 4], [1, 2, 3, 5] y [1, 2, 4, 5]. Esto se puede observar en la siguiente imagen:

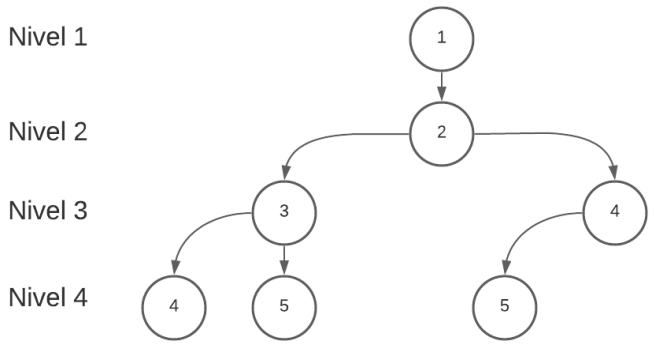

Figure 5: Basado en el árbol de la figura 3, esta seria la poda para $A = [1, 2, 3, 4, 5]$ y $n = 3$. Como se observa, todos los nodos del árbol deben pertenecer a A y el árbol solo llega hasta el nivel $n + 1 = 4$. De esta manera, las soluciones serian todos los caminos de las hojas del árbol en el nivel $n + 1$ hasta la raíz.

En caso de que árbol no se pueda expandir hasta el nivel $n + 1$, quiere decir que no existen soluciones. Esto puede suceder cuando los nodos intermedios que están en los niveles antes del $n + 1$, no están en A. Por ejemplo cualquier arreglo A que no tenga el 1, no tiene soluciones. De esta manera se propone el siguiente pseudocódigo que utiliza esta idea para resolver el problema:

```
SOLVE(A, n, k, sol):
  1) si k = 0 y 1 esta en A entonces add 1 a sol y SOLVE(A,n,1,sol)
  2) de lo contrario, si n = longitud sol, entonces imprimir sol
  3) de lo contrario, si n > longitud sol
    a) por cada valor i en sol
      +)si k + i esta en A
        -) add k + i a sol
        -) SOLVE(A,n,k+i,sol)
        -) quitar k + i de sol
```

4.2 Explicación del pseudocódigo:

En cuanto a los parámetros, n es el mismo de la entrada mas uno, A es el mismo arreglo de la entrada, k simbolizara el nodo donde me encuentro en el árbol de soluciones y sol el camino que he recorrido para llegar al nodo k. La idea es que cada vez que se llegue a un nodo de nivel $n + 1$, se imprima la solución que se tiene en ese instante. A continuación se describe cada caso:

- Cuando $k = 0$ quiere decir que se esta empezando, por lo cual se verifica que el 1 este en A y en caso de estarlo se añade a sol y se continua expandiendo el árbol hasta el nivel $n + 1$.

- Cuando n = longitud de sol, quiere decir que se ha llegado al nivel $n+1$, por lo cual se imprime la solución que se lleva hasta el momento
- Finalmente, si la solución todavía no tiene la longitud $n+1$, entonces me encuentro en un nodo k intermedio, por lo cual genero todos los hijos de ese nodo, que necesariamente estén en A. Luego utilizando la estrategia de backracking, recorro los hijos en busca de las soluciones con SOLVE y después me devuelvo en el árbol para generar los demás hijos de otros k intermedios.

De esta forma, se puede cambiar la salida del algoritmo:

Nueva salida: SOLVE(A, n+1, 0, [ ]).

4.3 Implementación en Python:

A continuación se presenta una implementación en Python del algoritmo descrito:

```
def solve(A, n, k, sol):
  if(k == 0 and 1 in A):
    sol.append(1)
    solve(A, n, 1, sol)
  elif(n == len(sol)): print(sol)
  elif(n > len(sol)):
    for i in sol:
      if(k + i in A):
        sol.append(k + i)
        solve(A, n, k + i, sol)
        sol.pop()
```