Micael Andrade Dos Santos

1. Questão

   a. `Descreva Sua ideia de solução`

   Modelei o problema de tal forma que os dígitos ficam em uma matriz. Ou seja, para os dígitos 110 121 132 143 temos a seguinte representação

   $[[1,1,0],[1,2,1],[1,3,2],[1,4,3]]$.

   Dividi o problema em três funções. `K_backtracking` responsável por gerar todas as combinações possíveis para o número de n, `filterValidSum` que verifica se um determinado dígito é válido ou não e `k_digits` responsável por empacotar às duas funções anteriores e retornar todos os dígitos válidos.

   b. `Elabore o algoritmo em pseudo-linguagem`

```
algoritmo k_digits(n)
{--
ENTRADA: Um inteiro representando o tamanho do k dígito, inteiro i iniciando em 0 e uma lista c
representando uma possível combinação.

SAÍDA: Uma matriz contendo os possíveis k_digítos, tal que a soma
dos elementos em posições pares é iagual a soma dos elementos nas
posições ímapres. Ex: para n=3 -> [[1, 1, 0], [1, 2, 1], [1, 3, 2]...]
--}
inicio
    combinacoes := inicializaVetor(0,n)--Inicializa um vetor de tamanho n com todos valores 0
    matrix_k_digits := inicializaMatriz(10^n) --Aloca mémoria para uma matriz com todos k_digitos.

    K_backtracking(n, 0, combinacoes, matrix_k_digits)
    filtre(validSum, matrix_k_digits) --aplicando a função ValidSum para retornar apenas os dígitos válidos
fim
```

```
algoritmo K_backtracking(n, i, c, m)
{-
   ENTRADA: Inteiro n representando a quantidade de dígitos.
   inteiro i inicialmente como 0, lista de combicações c, matrix m
   onde será armazenado todos k_dígitos.
-}
inicio
  se(i = n) então
    m.insira(c) -- Inserindo um novo dígito em minha matriz de dígitos
  senão então
    para j=0  até 9 faça -- gerando novas combinações
      inicio
        c[i] = j -- nova combinação
        K_backtracking(n, i+1, c, m) -- Realizando o backtracking
      fim
fim
```

```
algoritmo validSum(l)
{-
   ENTRADA: Uma lista de dígitos no formato [n1, n2, n3 ... nk]
   SAÍDA: False caso n1 seja 0 ou a soma dos itens nas posições pares forem diferentes da soma das posições ímpares.
-}
inicio
  soma_pares := 0
  soma_impares := 0

  se(l[0] == 0) retorne False
```

```
  para i=1 l.tamanho faça
    inicio
      se i mod 2 = 0 -- inídice par
          soma_pares += l[i]
      senão -- índice ímpar
          soma_impares += l[i]
    fim
  retorne soma_impares = soma_pares
fim
```

---

2. Questão

   a. `Descreva sua ideia de solução`

   Podemos definir qual elemento entra ou não no conjunto potência com combinações de 0s e 1s da seguinte forma.

   por exemplo, o conjunto `A={a,b}` temos as seguintes combinações possíveis. `P(A) = { [0,0], [1,0], [0,1], [1,1] }` Isso equivale a dizer:

   `P(A) = {[], [a], [b], [a,b]}`. Dessa forma, quando tivermos as possíveis combinações só é preciso varre-las para determinar quem entra ou não no conjunto potência.

   b. `Elabore o algoritmo em pseudo-linguagem`

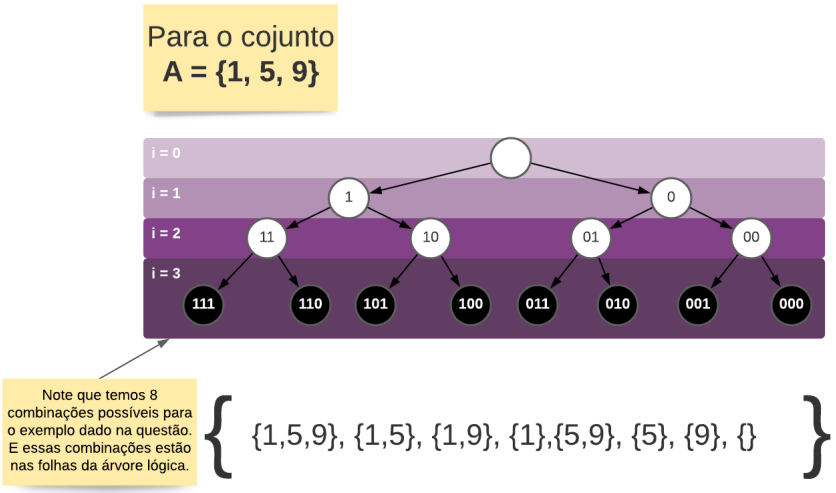
```
algoritmo conjuntoPotencia(C, n)
{-
  ENTRADA: Conjunto C de elementos com n valores.
  SAÍDA: Conjunto P potência de tamanho 2^n
-}
inicio
  B --Vetor de tamanho n com todas posições igual à 0
  matriz_comb =: inicializaMatrizVazia() -- Matriz de combinações possíveiscomo mostrado no item a)
  conjunto_pot := inicializaMatrizVazia() -- Matriz contendo P(C), ou seja, o conjunto potência.

  combicacoes(0, B, matriz_comb, n) -- Backtracking

  para j=1 até (2^n) faça
    inicio
      sub_conjunto := inicializaArryVazio()
      para i=1 até n faça
        inicio
          se(matriz_comb[j][i] = 1 então
            sub_conjunto.insira(C[i])
        fim
      conjunto_pot.insira(sub_conjunto)
    fim
  retorne conjunto_pot
-----------------------------------------------------------------------------------------
procedimento combinacoes(i, B, matriz_comb, N)
{-
ENTRADA:
Índice i do elemento no conjunto array B de combinações também de tamanho N
matriz_comb matriz onde estará as combinações.
-}
  se (i = N) então --Uma combinação possível
      matriz_comb.insira(B)
  senão então
      B[i] = 1 --Considera a inclusão do elemento C[i]
      combinacoes(i+1, B, matriz_comb, N) --Backtracking

      B[i] = 0 -- Não considera a inclusão do elemnto C[i]
      combinacoes(i+1, B, matriz_comb, N) --Backtracking
fim
```

c. `Árvore da Backtracking`



Árvore lógica para oconjunto dado na questão

---

4. Questão

1. `Descreva sua ideia de solução`

   Irei tentar resolver o problema utilizando o método probabilístico Las Vegas. Pois, precisamos de um valor exato para o problema.

   Como estamos utilizando o método Las Vegas, precisamos de uma função que realize a checagem dos dados, chamarei essa função de `check_array_valid` que será invocada várias vezes. Iremos sortear um `index` aleatório que pertence a meu espaço amostral, o elemento cujo `index` acabou de ser sorteado será inserido no meu vetor final de resultados. Caso sua inserção na solução não infrinja minha restrição, deixamos esse elemento como parte da solução, caso contrário ele volta para nosso espaço amostral.

b. `Elabore o algoritmo em pseudo-linguagem`

```
algoritmo check_array_valid(arr, n)
{--
  ENTRADA: Vetor arbitrário de quantidade .
  SAIDA: True caso não haja três elmentos com a mesma paridade consecutivamente
--}
inicio
  para i=0 até n faça
    inicio
      par  := 0
      impar:= 0
      subArr := [i ... 3+i] -- pegando um range de três elementos(fatia)
        para j = 1 até 3 faça
          inicio
            se(subArr[j] % 2 = 0) faça
                par := par + 1
            senão
                impar := impar + 1
          fim
          se(par = TOLERANCIA || impar = TOLERANCIA):
              retorne False
      fim
  retorne True
fim
```

```
algoritmo lasVegas(L, n)
{--
  ENTRADA: Vetor L ordenado com a mesma quandidade de elmentos ímpares e pares.
  SAIDA: Vetor R com os elementos de L tal que não haja três elementos consecutivos com a mesma paridade.
--}
inicio
  R := criaListaEncadeada() -- Uma lista encadeada para solução do problema.
  enquanto n >= 0 faça
    inicio
      index := random(1, n) -- gerando um index aleatório.
      elemento := L.removaPorIndex(index) -- Remove um determinado elemento por index.
      R.insira(elemento) -- Inserindo no inicio do vetor R o elemento removido de L
      se (not check_array_valid(R)) então
          elemento:=R.remova() -- Retira o elmento que acabou de ser inserido, pois não é uma solução válida.
          L.insira(elemento) -- Volta para meu espaço amostral
    fim
  retorne R
fim
```

---

5. Questão

`Descreva sua ideia de solução`

Podemos usar dois casos em que as probabilidades são iguais e tomar proveito disso. O primeiro caso seria `(0, 1) = 0.6 * 0.4 =  0.24` o segundo caso seria, `(1, 0) = 0.4*0.6 = 0.24`. Note que ambos tem probabilidade de 24%. Para valores do tipo `(1,1)` ou `(0, 0)` basta chamar a função `escolhaJusta()` recursivamente para cair em um dos casos em que as probabilidades são iguais.

`Elabore o algoritmo em pseudo-linguagem`

```
algoritmo escolha()
{--
  Função escolha Viciada
  SAIDA: Int 0 com 60% de probabilidade ou inteiro 1 com 40% de probabilidade.
--}
inicio
  valor = random(1, 10)
    se(valor <= 6)
        retorne 0
    senão
        retorne 1
fim
```

```
algoritmo escolhaJusta()
{-- ENTRADA: Dada uma função viciada.
    escolhaJusta retornara uma probabilidade de 50% para ambos os casos de
    escolha.
SAIDA: 0 ou 1 Representando os casos da escolha.
--}
  val1 := escolha()
  val2 := escolha()

  se(val1 = 0 && val2 = 1) então
      retorne 0
  se(val1 = 1 && val2 = 0) então
        retorne 1
  retorne escolhaJusta() -- Até cair em um dos casos acima
fim
```