Relatório do Trabalho de Ordenação e Estatísticas de Ordem

Técnicas de Programação Avançada — Ifes — Campus Serra

Alunos: Douglas Bolis,Jadson Pereira e Guilherme Bodart

Prof. Jefferson O. Andrade

29 de setembro de 2018

Sumário

Lista de Códigos Fonte

Lista de Figuras

1 Introdução

Este trabalho consiste na resolução de um conjunto de problemas envolvendo as estruturas de dados básicas vistas (ou revistas) na disciplina de Tópicos Avançados de Programação.

Todos os problemas propostos neste trabalho estão disponíveis no site UVa Online Judge.

2 Implementação do Trabalho

2.1 Warn Up

2.1.1 UVa 12247 – Jollo

O problema do Jollo é definir qual a menor carta que o servente tem que dar para o príncipe para ele ganhar da princesa, visto que ele sempre perde para a princesa e quando perde, ele chora, incomodando todos que estão próximo.

Analisando o Código Fonte 1,para solucionar esse problema foi feito a distribuição das cartas, utilizando uma lista para as cartas da princesa e uma lista para as cartas do príncipe (linha 6 a 8), onde essas listas são tratadas para serem uma lista de inteiros ao invés de string, para assim, ordena-las com o método *sort()* (linha 9 e 10). Com as listas ordenadas em mãos e utilizando uma lista de cartas usadas de 53 elementos para verificar se as cartas entregue ao príncipe estão disponíveis e uma flag que varia de 0 e 1 para validar se a operação já foi feita ou não.

Inicialmente, é desabilitado as cartas entregue para o príncipe e a princesa(linha 11 a 15), para não ocorrer o caso da resposta ser uma carta indisponível, a primeira analise feita (linha 16 a 21) é se as duas cartas do príncipe forem maiores que as cartas da princesa, ou seja, se a menor carta na lista ordenada do príncipe é maior que a maior carta da lista ordenada da princesa, nesse caso, qualquer carta disponível satisfaz e da a vitória ao príncipe. Outra analise(linha 22 a 30) é se apenas uma carta do príncipe for maior que todas as da princesa, nesse caso, como o jogo é uma melhor de três, caso o príncipe ganhe duas ele vença, com isso é feito a procura da menor carta disponível que seja maior que a maior carta da princesa representado pela variável *princess[2]*, nessa analise é feita também uma verificação se há cartas disponíveis, no caso de ser entregue as cartas: 50 e 51 para a princesa e 52 para o príncipe, nesse caso não há carta disponível no baralho para o príncipe ganhar.

Outra analise (linha 31 a 36) é se as cartas do Príncipe forem menor que só uma carta dela, nesse caso é feito uma procura da segunda maior carta da princesa. Por fim, é analisado(linha 38 a 41) se as cartas da princesa forem maiores que as duas cartas do príncipe, nesse caso, não tem como ele ganhar. Após isso, já é capaz de ter a resposta da carta certa, a função entrada é responsável apenas para ler os dados para fazer a analise.

#	Problem	Verdict	Language	Run Time	Submission Date
24173212	12247 Jollo	Accepted	PYTH3	0.010	2019-11-09 22:21:01

Figura 1: Mensagem de Aceitação do *UVa 12247 – Jollo*.

```
cartas = entrada()
avaliable = [1] * 53 # lst[53] cartas disponiveis
flag = 0                # Verifica se ja passou na função.
answer = 100
while(cartas != "0 0 0 0 0"):
    lstCartas = cartas.split(" ")
    princess = [int(lstCartas[0]),int(lstCartas[1]),int(lstCartas[2])]
    prince = [int(lstCartas[3]),int(lstCartas[4])]
    princess.sort()
    prince.sort()
    avaliable[princess[0]] = 0
    avaliable[princess[1]] = 0
    avaliable[princess[2]] = 0
    avaliable[prince[0]] = 0
    avaliable[prince[1]] = 0
    if(prince[0]>princess[2] and flag == 0):
        for i in range (1,len(avaliable)):
            if(avaliable[i]==1):
                answer = i
                flag = 1
                break
    if(prince[1] > princess[2] and flag == 0):
        i=princess[2] +1
        while(i<53):
            if(avaliable[i]==1):
                answer = i
                break
            i += 1
        if(i>52):
            answer = -1
    if(prince[0]>princess[1] and flag == 0):
        i=(princess[1]) + 1
        while(i<53):
            if(avaliable[i]==1):
                answer = i
                break
            i += 1
    if(prince[0]<princess[1] and prince[1] < princess[2] and flag == 0) :
        answer = -1
        avaliable[princess[2]] = 0
    avaliable[answer] = 0
    print(answer)
    #reseta a lista
    flag = 0
    avaliable = [1] * 53
    cartas = entrada()
```

Código Fonte 1: Solução do Jollo

2.2 Vetores

2.2.1 UVa 10038 – Jolly Jumpers

O problema do Jolly Jumpers é definir se uma sequência de números é ou não é Jolly. Para uma sequência ser Jolly ela precisa que os valores absolutos das diferenças entre elementos sucessivos sejam todos 1...n-1, por exemplo 1 4 2 3, que a diferença seria 3,2,1. Isso implica que qualquer sequência de um único número é Jolly. A entrada consiste em um número N, que será a quantidade de valores da sequência, por exemplo: 4 1 2 4 7, uma sequência válida.

Na solução foi feita uma lista de tamanho 3000, que é o máximo de número de entrada, a primeira condição para ver se é Jolly, é se o primeiro valor, é um e se só tem um número após ele, se sim, é Jolly. (No UVa não é necessário tratamento de erro). Depois disso é verificado para cada 2 números suas diferenças absolutas, e guardado 1 na listaSoma no respectivo valor, 2 ficaria na 3 posição da listaSoma, no caso listaSoma[2], e varia isso até acabar a sequência, no final teria uma sequência de 1 na posição 1 até N, ou não, caso tenha é Jolly, se não, não é Jolly, para verificar apenas somaria os valores da lista e verificar se deu N-1; Fizemos também uma verificação para o algoritmo não percorrer a lista de 3000 no final não importa o tamanho dela, ele verifica onde ta o primeiro 1, e depois para até encontrar o primeiro 0, tirando assim algumas verificações desnecessárias.

Um porém: O código não funciona sem o Try: except: inicial, antes de fazer o código estávamos conversando com um colega, Andreas, e ele disse que fez o código e ficava dando erro no UVa de tempo de execução, e depois de procurar achou um código com o try: except: então colocou no código e funcionou, então já tínhamos uma ideia de que ia precisar disso, o motivo de precisar é porque no enunciado não tem uma condição de parada, então precisa dele, em Python, em algumas outras linguagens pode ser que não seja necessário.

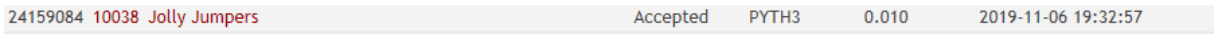

Figura 2: Mensagem de Aceitação do *UVa 10038 – Jolly Jumpers*.

```python
while True:
    # Até terminar as entradas
    try:
        lista = list(map(int, input().split()))
    except EOFError:
        break
    #Lista que conterá os valores para verificar o Jolly
    listaSoma = [0]*3000
    #Verifica a entrada simples : 1 2000
    if (len(lista)==2 and lista[0]==1):
        print("Jolly")
    else:
        #Coloca na listaSoma 1 em cada posicao de valor igual a sequencia para o Jolly
        for i in range(1,len(lista)-1):
            valor = abs(lista[i] - lista[i+1])
            #Condicao para adionar na Soma
            if valor < lista[0] and listaSoma[valor] == 0:
                listaSoma[valor] = 1
        i = 0
        achoum = False
        soma = 0
        #Verifica sem a sequencia tem 1, se não não é Jolly
        if(listaSoma[1]==1):
            while True and i < 3000:
                soma = soma + listaSoma[i]

                #Encontra o primeiro 1 depois do 0;
                if listaSoma[i]==1 and not achoum:
                    achoum = True
                #Um meio de sair do loop antes do final, caso encontre um 0 antes do tamanho final, 3000;
                if achoum:
                    if listaSoma[i]==0:
                        break
                i = i + 1
            #Verificacao do Jolly
            if soma == lista[0] - 1:
                print("Jolly")
            else:
                print("Not jolly")
        else:
            print("Not jolly")
```

Código Fonte 2: Solução para o problema Jolly Jumpers

2.2.2 UVa 11340 – Newspaper

O problema *Newspaper* consiste no cálculo do valor a ser pago(em dinheiro) por uma Agência de Notícias aos autores dos artigos que serão publicados pela Agência de Notícias.

Para a Agência de Notícias pagar pelos artigos ela leva em conta algumas regras e ela possui uma tabela de valores contendo os caracteres com os valores(em centavos) de cada caracter, alguns caracteres podem ter valor igual a zero. O cálculo do valor é realizado no artigo em sua totalidade e o autor do artigo recebe seu pagamento como uma soma dos valores de todos os caracteres que aparecem no artigo e que estão na tabela de valores da Agência.

Analisando o Código Fonte 3 temos a implementação para o problema *UVa 11340 – Newspaper*. Nela foi implementada utilizando um vetor com tamanho 128 que para armazenar todos os valores dos caracteres referente à Tabela ASCII(linha 21).

Inicialmente para todos os caracteres são setados com -1 para indicar que o caracter não possui um valor definido pela agência, e em seguida é lido a tabela de caracteres definido pela agência e armazenado o valor de cada caracter na posição do seu código ASCII no vetor.

Para calcular o valor a ser pago ao autor do artigo é realizado a leitura do artigo linha por linha, e para cada caracter lido do artigo é verificado se o mesmo possui um valor definido pela agência na tabela de valores.

E ao final de cada artigo é imprimido como saída o valor a ser pago ao autor pleo artigo escrito e publicado pela Agência de Notícias.

Figura 3: Mensagem de Aceitação do *UVa 11340 – Newspaper*.

```java
import java.util.Scanner;

public class Main {
    public final static int SIZE_ASCII_TABLE = 128;

    public static void main(String[] args) {
        try {
            Scanner in = new Scanner(System.in);
            int[] tableCosts = new int[SIZE_ASCII_TABLE];
            int numberTests = in.nextInt();

            for (int n = 0; n < numberTests; n++) {
                int numberCharacters = in.nextInt();
                int finalCost = 0;

                for (int i = 0; i < SIZE_ASCII_TABLE; i++) {
                    tableCosts[i] = -1;
                }

                for (int i = 0; i < numberCharacters; i++) {
                    int charKey = (int) in.next().charAt(0);
                    int charCost = in.nextInt();
                    tableCosts[charKey] = charCost;
                }

                int numberLinesArticle = in.nextInt();
                in.nextLine();

                for (int j = 0; j < numberLinesArticle; j++) {
                    String lineArticle = in.nextLine();
                    for (int k = 0; k < lineArticle.length(); k++) {
                        int charKey = (int) lineArticle.charAt(k);
                        finalCost += (charKey < 0 || charKey >=
                        ↪  SIZE_ASCII_TABLE || tableCosts[charKey] < 0) ? 0 :
                        ↪  tableCosts[charKey];
                    }
                }
                System.out.println(String.format((finalCost % 100 < 10) ?
                ↪  "%d.0%d$" : "%d.%d$", finalCost / 100, finalCost % 100));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.exit(0);
        }
    }
}
```

Código Fonte 3: Solução para o problema *Newspaper*

2.3 Matrizes

2.3.1 UVa 11581 – Grid successors

Nesse exercício, pelo enunciado foi difícil entender o que tinha que ser feito, para entender melhor, procuramos alguém na internet que teria feito e explicado o que é para ser feito, depois disso percebemos como o código era simples.

O problema do Grid successors, é definir quantas vezes é necessária usar a função, que chamamos de grid(), em uma matriz, no caso 3x3, para que todos os valores se torne 0, retorna -1 se não fizer nenhum movimento, e n-1 movimentos se fizer algum movimento.

A solução começa escrevendo a matriz de entrada, antes de chamar a função tem um while que verifca se toda a matriz é zero, se é acaba, se não chama a função, e repete até tudo ser zero. A função grid(), ela calcula novamente os valores da matriz utilizando os vizinhos de cada cédula, no caso a cédula [0][0], tem os vizinhos, [0][1] e [1][0], os vizinhos são aqueles que estão ou a esquerda, direita, ou em cima ou baixo. O cálculo consiste em somar os valores do vizinho e pegar o resto da divisão por 2. Fazendo isso para todas as cédulas cada vez que a função é chamada, a matriz será sempre diferente, e pelo que da para perceber devido aos testes, o máximo de tentativas que ocorreu foi 4, resposta 3, só precisa passar na função no máximo 4 vezes para a matriz 3x3 ser zerada, em qualquer caso.

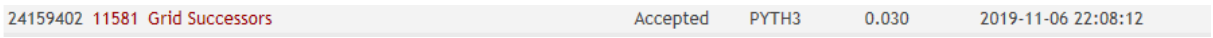

Figura 4: Mensagem de Aceitação do *UVa 11581 – Grid successors*.

```python
def tudoZero(matriz):
    for i in range(3):
        for j in range(3):
            if matriz[i][j]!=0:
                return False
    return True

#Calcula o grid de cada cedula da matriz usando seus vizinhos (soma dos vizinhos)%2
def grid(matriz):
    matrizG = [[0,0,0],[0,0,0],[0,0,0]]
    #Cada caso possivel com seus vizinhos
    matrizG[0][0] = (matriz[0][1]+matriz[1][0])%2
    matrizG[0][1] = (matriz[0][0]+matriz[1][1]+matriz[0][2])%2
    matrizG[0][2] = (matriz[0][1]+matriz[1][2])%2
    matrizG[1][0] = (matriz[0][0]+matriz[1][1]+matriz[2][0])%2
    matrizG[1][1] = (matriz[0][1]+matriz[1][0]+matriz[1][2]+matriz[2][1])%2
    matrizG[1][2] = (matriz[1][1]+matriz[0][2]+matriz[2][2])%2
    matrizG[2][0] = (matriz[1][0]+matriz[2][1])%2
    matrizG[2][1] = (matriz[2][0]+matriz[1][1]+matriz[2][2])%2
    matrizG[2][2] = (matriz[2][1]+matriz[1][2])%2

    return matrizG

def main():
    #inicia matriz
    matriz = [[0,0,0],[0,0,0],[0,0,0]]
    num = int(input())
    for n in range(num):
        for i in range(3):
            string = input()
            if(string==''): #Pula linha lida vazia
                string = input() #Pega uma string de tamanho 3
            for j in range(3):
                matriz[i][j] = int(string[j]) #Valores da matriz sendo preenchidos


        contador = -1
        #While que verifica quantas vezes a funcao grid foi usada até todos os valores ficarem 0
        while(not tudoZero(matriz)):
            matriz = grid(matriz)
            contador = contador + 1
        print(contador)
    return 0
```

Código Fonte 4: Solução para o problema do Grid Sucessors

2.4 Ordenação

2.4.1 UVa 10107 – What is the Median?

O problema consiste em achar a mediana de uma lista, para achar a mediana de uma lista é necessário que a lista esteja ordenada, a mediana é o elemento que fica no meio da lista, por exemplo: [1,3,5,7,9], o 5 é a mediana, mas no caso da lista ter tamanho par, [2,4,6,8], a mediana é os dois do meio somado sobre dois, nesse caso a mediana é 5. O problema consiste em várias entradas de dados sendo números, estes números entraram na lista, e uma nova mediana é calculada e apresentada toda vez, até acabar a entrada, sendo N o total de números N ¡ 100000.

A solução é bem simples, entra um número, coloca na lista, ordena a lista e depois calcula a mediana, se for ímpar só divide por 2 o tamanho da lista, e pega o elemento central, se for par, divide por 2 o tamanho da lista, e pega os dois elementos do meio, soma eles e divide por 2.

Neste código também foi necessário colocar um try: except: para a entrada de dados, mas neste caso foi por não colocar um limitador de até 100000 entradas, achamos melhor colocar deste jeito, para caso tenha entradas de N's menores.

Figura 5: Mensagem de Aceitação do *UVa 10107 – What is the Median?*.

```
lista = []
while True:
    # Até terminar as entradas
    try:
        num = int(input())
    except EOFError:
        break

    lista.append(num)
    #Ordena lista
    lista.sort()
    #Tamanho Ímpar
    if(len(lista)%2==1):
        print(lista[len(lista)//2])
    #Tamanho Par
    else:
        print(((lista[(len(lista)//2)-1] + lista[len(lista)//2]))//2)
```

Código Fonte 5: Solução para o problema Whats is the median?

2.5 Manipulação de bits

2.5.1 UVa 10264 – The Most Potent Corner

Este problema tem como objetivo achar a maior potência de dois cantos vizinhos, sendo que os pesos de cada canto é dado, terá que calcular a soma máxima dos pesos de dois vizinhos.

A entrada consiste em um primeiro número dizendo qual será o tamanho do cubo, e depois uma quantidade de 2^3 números dizendo o peso de cada canto do n-cubo. Sendo que os pesos são respectivamente do 00...00, 00...01, 00...10 e assim por diante. O resultado deve mostrar apenas a maior soma dos maiores cantos, vizinhos.

Este foi um problema um pouco complicado de entender exatamente o que era pedido, estava pensando que era para calcular a soma de todos os vizinhos de uma matriz quadrada, o que já foi pedido em outro exercício, então aconteceu muitos testes de mão até achar a ideia do exercício, e mais um pouco até entender que era pra ser a soma dos vizinhos de dois vizinhos.

Como a imagem abaixo um cubo de N sendo 3, seria neste formato. Os vizinhos de um canto são aqueles onde é alterado apenas 1 bit do cubo, 110 é vizinho de 010-111-100. Depois que percebi isso e que tinha que somar a soma dos vizinhos de 1 canto mais a soma dos vizinhos de outro canto, sendo este vizinho do primeiro canto, ficou muito mais claro resolver o problema.

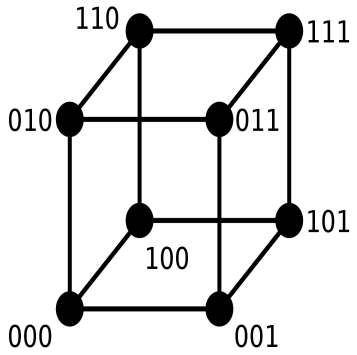

Figura 6: Imagem explicativa *UVa 10264 – Cubo Binário.*

Em um loop eterno, começa perguntando o tamanho do n-cubo, é então criado dois dicionários, um que guardará os pesos dos cantos e outro para guarda a soma dos vizinhos de cada canto, e uma lista, que me dirá quem são os vizinhos de cada canto, servirá para pesquisar e somar os valores e achar a maior soma.

Começo lendo todos os números do n-cubo e colocando-os no dicionário com suas chaves 00..01,00.10 etc. Depois para cada canto eu pesquiso os vizinhos, o IF da linha 20, e o ELSE da linha 25, são as partes mais importantes, onde foi feito uma maneira para achar cada vizinho, é lido a chave, uma string, letra por letra, perguntando se é 1 ou 0, se for 0 cria uma nova string com 1 no local, ou o contrário, verifica se a combinação existe no cubo, caso exista atribui a soma e a lista de vizinhos, e no final de cada canto é colocado a soma no valor do dicionário da soma com a mesma chave do canto.

Ao final de tudo verifica qual a maior soma que existe entre dois vizinhos, e mostra ela na saída de cada caso. O código foi modificado nas linhas 6 e 29 apenas para cabimento do código na página. Pule as linhas no lugar dos ";".

24208704	10264 The Most Potent Corner	Accepted	PYTH3	0.580	2019-11-17 20:46:33
24206651	10264 The Most Potent Corner	Time limit exceeded	PYTH3	3.000	2019-11-17 12:19:45

Figura 7: Mensagem de Aceitação do *UVa 10264 – The Most Potent Corner*.

```
while(True):
    try:
        num = int(input())
    except EOFError:
        break
    cubo = {}; cuboSoma = {}; listaVizinhos = []
    for i in range((2**num)):
        try:
            peso = int(input())
        except EOFError:
            break
        numB = bin(i)[2:]
        chave = str(0)*(num-len(numB)) + numB
        cubo[chave] = peso
    for key in cubo:
        soma = 0
        for k in range(num):
            if(key[k])=='0':
                string = key[0:k]+ '1' + key[k+1:]
                if string in cubo:
                    soma = soma + cubo[string]
                    listaVizinhos.append(string)
            else:
                string = key[0:k]+ '0' + key[k+1:]
                if string in cubo:
                    soma = soma + cubo[string]
                    listaVizinhos.append(string)
        cuboSoma[key] = soma
    maior = 0; y = 0
    for s in cuboSoma:
        for e in range(num):
            try:
                soma = cuboSoma[s] + cuboSoma[str(listaVizinhos[(y*num + e)])]
            except:
                soma = cuboSoma[s]
            if(soma>maior):
                maior = soma
        y = y + 1
    print(maior)
```

Código Fonte 6: Solução para o problema The Most Potent Corner

2.5.2 UVa 11926 – Multitasking

O problema consiste em uma agenda dos próximos um milhão de minutos onde tem que ver se nas atividades propostas terá conflito ou não.

A entrada começa com dois números, N e M, sendo N as atividades que não se repetem e M atividades que se repetem, a entrada de N é um tempo inicial e um tempo final da atividade, até N atividades. E M é um tempo inicial, um tempo final e de quanto em quanto tempo ele se repete, até M atividades.

Sendo que se repete até N e M sejam ambos 0;

A solução para o problema foi uma while que é verdade até que N e M sejam ambos zeros, usando o bitset, listaBit, do C++ de tamanho 1000005, e um booleano para verificar se houve ou não conflito.

Depois da entrada de N e M, começa um for que vai dos inicio até os fins de cada atividade N, preenchendo para 1 a listaBit caso não tenha nada naquele intervalo, e caso tenha, termina o for e "conflite"se torna true.

Um outro for para verificar as atividades M, um while que vai até um milhão ou enquanto não há conflito, então caso o primeiro for já tenha dado algum conflito, não entrará no while. O for dentro do while para ir andando na listaBit verificando se tem algo nos intervalos e preenchendo 1, caso tenha, termina o for e "conflite"se torna true, sendo que o for vai de inicio ao final, mas o pulo dele é do tamanho da variável "repeticao", e no final se tem ou não conflito é mostrado.

24186224 11926 Multitasking	Accepted	C++11	0.010	2019-11-12 18:03:25
24183927 11926 Multitasking	Time limit exceeded	PYTH3	1.000	2019-11-12 11:21:41

Figura 8: Mensagem de Aceitação do *UVa 11926 – Multitasking*.

```cpp
#include <cstdio>
#include <bitset>
using namespace std;


int main(){
    while(true){
        int n,m;
        scanf("%d",&n);
        scanf("%d",&m);
        if(n==0 && m==0){break;}
        bitset<1000005> listaBit;
        bool conflite = false;
        for(int q=0;q<n;q++){
            int inicio,fim;
            scanf("%d",&inicio);
            scanf("%d",&fim);
            for(int i=inicio;i<fim;i++){
                if(!listaBit.test(i)){listaBit.set(i);}
                else{
                    conflite=true;
                    break;
                }
            }
        }
        for(int k=0;k<m;k++){
            int inicio,fim,repeticao;
            scanf("%d",&inicio);
            scanf("%d",&fim);
            scanf("%d",&repeticao);
            while(!conflite and inicio < 1000000){
                for(int j=inicio;j<fim;j++){
                    if(!listaBit.test(j)){listaBit.set(j);}
                    else{
                        conflite = true;
                        break;
                    }
                }
                inicio = inicio + repeticao;
                fim = min(fim + repeticao,1000000);
            }
        }
        if(conflite){printf("CONFLICT\n");}
        else{printf("NO CONFLICT\n");}
    }
}
```

Código Fonte 7: Solução para o problema Multitasking

2.6 Lista Encadeada

2.6.1 UVa 11988 – Broken Keyboard

O problema do *broken keyboard* se refere a um teclado parcialmente quebrado, que as vezes a tecla *Home*('[') ou a tecla *end*(']') são pressionadas automaticamente, focado no texto e com monitor desligado, só é possível ver a mensagem (*Beiju text*) quando termina de escrever e liga o monitor.

Analisando o Código Fonte 8, a solução para fazer um *Beiju Text* foi inicialmente inicializar uma linked list (lista simplesmente encadeada) vazia e ir adicionando os elementos com auxilio de uma flag que nos indica se o botão home esta apertado ou não (linha 21 a 27), caso ele esteja (flag = 1), é adicionado no inicio os valores em sequência, dando um 'shift' para a direita nos que já estão na lista,isso é feito com o auxilio da variável iHome, que é o index de onde deve ser inserido as variáveis, caso a flag esteja zerada, ou seja o botão home não esta apertado, a letra é adicionada no final.

"O principal benefício de uma linked list a um array convencional é que os elementos da lista podem ser facilmente inseridos ou removidos sem realocação ou reorganização de toda a estrutura, porque os itens de dados não precisam ser armazenados contiguamente na memória ou no disco, enquanto a reestruturação de um array é feita. o tempo de execução é uma operação muito mais cara. As Linked list permitem a inserção e remoção de nós em qualquer ponto da lista e o fazem com um número constante de operações, mantendo o link anterior ao link sendo adicionado ou removido na memória durante o percurso da lista."Linked List

Vale ressaltar que, a *Linked list* possui vários métodos já implementados, usando o método 'add' sem passar nenhum index, o elemento é adicionado no final da lista, poderia ter usado o método addFirst para adicionar no inicio,porém como é passado uma série de valores após, esse método fica inviável, por isso optamos pelo 'add' passando um index(no caso o iHome), que por sua vez adiciona um elemento em uma lista na posição desejada e faz um 'shift' no elemento que estava antes nessa posição para direita.

#	Problem	Verdict	Language	Run Time	Submission Date
24181060	11988 Broken Keyboard (a.k.a. Beiju Text)	Accepted	JAVA	0.650	2019-11-11 16:52:02

Figura 9: Mensagem de Aceitação do *UVa 11988 – Broken Keyboard.*

```java
//package uva11988_brokenkeyboard;

import java.io.IOException;
import java.util.LinkedList;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) throws IOException {
        Scanner scan = new Scanner(System.in);
        LinkedList<Character> list = new LinkedList();
        int flag;
        int iHome;
        while(scan.hasNext()) {
            list.clear();
            flag = 0;
            iHome = 0;
            String line = scan.nextLine();
            for(int i=0;i<line.length();i++){
                switch (line.charAt(i)) {
                    case '[':
                        flag = 1;
                        iHome = 0;
                        break;
                    case ']':
                        flag = 0;
                        break;
                    default:
                        if(flag == 0){
                            list.add(line.charAt(i));
                            iHome = 0;
                        }else{
                            list.add(iHome,line.charAt(i));
                            iHome++;
                        }
                        break;
                }

            }
            StringBuilder s = new StringBuilder();
            list.forEach(c->{
                s.append(c);
            });
            System.out.println(s);
        }
    scan.close();
    }

}
```

Código Fonte 8: Solução para o problema do broken keyboard

2.7 Pilhas

2.7.1 UVa 00514 – Rails

O problema consiste em trens que chegam da estação A e querem ir para estação B, porém em A o trem chega com os vagões ordenados de 1,2...,N. Então o chefe precisa para o trem para B, porém ordenados em a1,a2,a3...,a. Que é definido na entrada de dados.

Por exemplo, a entrada é 5 4 3 2 1, o trem chega 1 2 3 4 5, e tem que sair para B na ordem 5 4 3 2 1. Para isso tem algumas regras, o trem que chega em A, antes de ir para B, para por uma estação, seus vagões podem se separar individualmente, sempre que um vagão entrar na estação, não pode mais voltar para A, e sempre que um vagão ir para B, não pode voltar para a estação, sendo que pode ter quantos vagões forem necessários na estação.

Foi um problema de difícil entendimento, já que foi difícil perceber que o trem chega em A sempre como 1...N, e sai para B, na sequência que foi dada como entrada.

A solução foi um pouco difícil pensar, tive que desenhar para ver o que seria o melhor jeito de fazer, o trem chega na sequencia, por exemplo, [1, 2 ,3], então foi feito uma variável E, para ser com o trem que entra na estação pelo trilho A.

Começa verificando se a pilha está vazia, se não limpa ela, este caso é mais para depois da primeira interação, em testes que deram resposta errada, a pilha fica com resto. Temos um for que vai de 0 a NUM, que é a quantidade de vagões, ele só ira rodar quando 1 vagão ir para o trilho B.

Pega a primeira entrada, por exemplo [2,1,3],a primeira entrada seria 2, coloca numa lista entrada, e a partir dai ele procura o 2 nos vagões do trilho A. A primeira verificação é ver se tem alguma coisa na estação, que é a pilha do problema, e se o que tem no topo da pilha é igual a entrada, no caso 2, se algum dos casos forem falso, ele coloca na estação o vagão que tem no trilho A.

E repete essa verificação até achar o número do vagão igual o numero da entrada, quando achar, coloca o vagão numa lista de saída e tira ele da estação,(é como se passasse o vagão para o trilho B praticamente), e então faz tudo de novo, até a todos passarem para o trilho B, no final de tudo, pergunta se a lista de entrada é igual a lista de saída, se for, a resposta é correta, se não, a resposta é errada.

Para acabar mudar o tamanho do trem, é colocado um 0 na entrada da station(), e para acabar o problema, é colocado um 0 novamente na entrada da main().

Uma coisa que aconteceu, que tivemos que procurar na internet, é a saida, que é necessário ter um "contrabarra N"no final da resposta "Yes/No"e ainda um "contrabarra N"depois que acaba o programa, lemos isso em um fórum e vimos uma solução que tinha esses dois "contrabarra N", caso contrário o UVa não aceita.

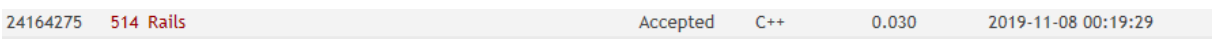

Figura 10: Mensagem de Aceitação do *UVa 00514 – Rails*.

```cpp
#include <cstdio>
#include <stack>
using namespace std;
void station(int num){
  int numVagao;
  stack<int> vagoes;
  int entrada[num];
  int saida[num];
  while(1){
      //Caso tenha algo na pilha
      while(vagoes.size()>0) {vagoes.pop();}
      int e = 0;
      for (int i=0;i<num;i++){
          scanf("%d",&numVagao);
          entrada[i] = numVagao;
          //Se colocar 0, volta a perguntar o tamanho do trem
          if(numVagao==0) {return;}
          while(e<=num){
              //Pilha nao vazia e se topo é igual a entrada
              if(vagoes.size()>0 && vagoes.top()==numVagao) {break;}
              //Da push
              vagoes.push(e+1);
              e = e + 1;
          }
          //Tira da pilha e coloca na lista de saida
          if(vagoes.top()==numVagao){
              saida[i] = numVagao;
              vagoes.pop();
          }
      }
      //Compara lista saida com a lista de entrada
      int verif = 0;
      for(int q=0;q<num;q++){

          if(!(saida[q]==entrada[q])){
              verif = 0;
              break;
          }
          else{verif=1;}
      }
      if (verif) {printf("Yes\n");}
      else {printf("No\n");}
    }
}
int main() {
  //Le ate ser 0 e para quando zerar
  while(1){
    int num;
    scanf("%d",&num);
    if(num==0) {break;}
    station(num);
    printf("\n");
  }
}
```


Código Fonte 9: Solução para o problema dos Trilhos

2.7.2 UVa 01062 – Containers

O terminal de contêineres precisa de um plano de empilhamento de contêineres para diminuir o tempo de carregamento. O plano deve permitir que cada navio seja carregado acessando apenas os contêineres mais altos das pilhas, e minimizando o número total de pilhas necessárias.

Para esse problema, sabemos a ordem em que os navios devem ser carregados e a ordem em que os contêineres chegar. Cada navio é representado por uma letra maiúscula entre A e Z (inclusive), e os navios serão carregado em ordem alfabética. Cada contêiner é rotulado com uma letra maiúscula representando o navio no que precisa ser carregado. Não há limite para o número de contêineres que podem ser colocados em um pilha única.

Analisando Código Fonte 10, quando chega um navio, é feito uma busca a procura do local ideal para depositar o container, essa busca é feita olhando o topo de cada pilha e comparando com o container do navio atual, visto na linha 16 ao 20, caso ache - o container é adicionado no topo dessa pilha, os containers só podem ser empilhados caso o container tenha um valor menor ou igual que o container do topo da pilha, caso não ache - é criado uma nova pilha e adicionado o container, depois adicionado essa nova pilha na lista de pilhas, visto na linha 23 até a linha 25.

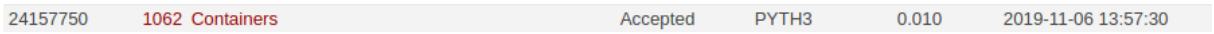

Figura 11: Mensagem de Aceitação do *UVa 01062 – Containers*.

```
#Leitura dos dados do arquivo.
def entrada():
    return input()
#inicio da solução
indexCase: int = 1
data = entrada()
while(data != 'end'):
    stackContainers:list = []
    if(data != ' '):
        #Percorrer a linha.
        for i in range(0,len(data)):
            currentContainer = data[i]
            if(currentContainer != '\n'):
                #Procura o container.
                j:int = 0
                while(j<len(stackContainers)):
                    if(currentContainer <= stackContainers[j][-1]):
                        stackContainers[j].append(currentContainer)
                        break
                    j +=1
                #endwhile
                #Não achou o container.
                if(j == len(stackContainers)):
                    stack = [currentContainer]
                    stackContainers.append(stack)
        #endfor
    #print(stackContainers)
    print("Case "+str(indexCase)+ ": "+str(len(stackContainers)))
    indexCase = indexCase + 1
    data = entrada()
#endwhile

  }
```

Código Fonte 10: Solução para o problema dos containers

2.8 Filas

2.8.1 UVa 10172 – The Lonesome Cargo Distributor

O problema *The Lonesome Cargo Distributor* consiste no cálculo de tempo em minutos no transporte de cargas dentro da intalação da *Airport Based Cargo Distribution Embarkation Facility(ABCDEF)* no Aeroporto Internacional de Cusco. Os aviões de carga chegam e partem para muitos países em todo o mundo e a ABCDEF tem como objetivo distribuir, carregar e descarregar as cargas transportadas por esses aviões.

Cada país X tem sua própria estação de carga identificada por seu ID de país. Existem duas plataformas(plataforma A e plataforma B) em cada estação. Na plataforma A, são colocadas as cargas que serão transportadas(pelo ar) para o país X em algum momento conveniente. A plataforma B é na verdade uma fila de cargas que devem ser transportadas para países que não sejam o país X.

Analisando o Código Fonte 11 temos a implementação para o problema *The Lonesome Cargo Distributor*. Nela foi implementada para identificar os conjuntos de trabalhos que serão realizados para os transporte de cargas para os diversos países*(linha 18 da primeira página de código)*. Para cada conjunto de trabalho é identificado e armazenado o número de estações circulares com cargas enfileiradas, o número de cargas que a transportadora carrega por vez e o número máximo de cargas que a fila na plataforma B pode acomodar. E para cada fila de cargas é indicado o número de cargas que a fila possui.

Para o cálculo do tempo total de trabalho de cada conjunto de trabalho implementamos uma fila para representar a plataforma B e uma pilha para representar a transportadora, definimos um *loop* que realiza a verificação se a instalação ou a transportadores estão vazias(*linha 13 da segunda página*) e enquanto esse é executado a plataforma B recebe a estação atual do processamento(*linha 16 da segunda página*). Implementamos um outro *loop* para realizar a descarga da transportadora na plataforma B. Ao final da execução de cada trabalho é imprimido o tempo total de trabalho.

#	Problem	Verdict	Language	Run Time	Submission Date
24208818	10172 The Lonesome Cargo Distributor	Accepted	JAVA	0.460	2019-11-17 22:21:53

Figura 12: Mensagem de Aceitação do *UVa 10172 – The Lonesome Cargo Distributor*.

```java
import java.io.IOException;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.Queue;
import java.util.Scanner;
import java.util.Stack;

class Main {
    private final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) throws IOException {
        Main main = new Main();
        main.run();
    }

    public void run() {
        int SET = scanner.nextInt();
        int N, S, Q;

        while(SET > 0) {
            N = scanner.nextInt();
            S = scanner.nextInt();
            Q = scanner.nextInt();

            LonesomeCargoDistributor cargo = new LonesomeCargoDistributor(N, S,
            ↪  Q);
            cargo.solve();
            SET--;
        }
    }

    class LonesomeCargoDistributor {
        Map<Integer, Queue<Integer>> facility;
        int N, S, Q, qi;

        LonesomeCargoDistributor(int N, int S, int Q) {
            facility = new HashMap<>();
            this.N = N;
            this.S = S;
            this.Q = Q;
        }

        public void solve() {
            for(int i = 1; i <= N; i++) {
                facility.put(i, new LinkedList());
                qi = scanner.nextInt();

                for(int j = 0; j < qi; j++) {
                    facility.get(i).add(scanner.nextInt());
                }
```

```
            }
            System.out.println(solveTotalMinutesJob());
        }

        public int solveTotalMinutesJob() {
            Queue<Integer> platB;
            Stack<Integer> carrier = new Stack<>();

            int minutes = 0;
            int i = 0;
            int cargo;

            while(!facilityIsEmpty() || !carrier.empty()) {
                i = (i % N) + 1;
                minutes += 2;
                platB = facility.get(i);

                while(!carrier.empty() && (carrier.peek() == i || platB.size()
                ↪  < Q)) {
                    cargo = carrier.pop();
                    if (cargo != i) {
                        platB.add(cargo);
                    }
                    minutes++;
                }

                while (platB.size() > 0 && carrier.size() < S) {
                    carrier.push(platB.poll());
                    minutes++;
                }
            }
            return ((minutes == 0) ? minutes : (minutes - 2));
        }

        public boolean facilityIsEmpty() {
            for(int i = 1; i <= N; i++) {
                if (facility.get(i).size() > 0) {
                    return false;
                }
            }
            return true;
        }
    }
}
```

Código Fonte 11: Solução para o problema *The Lonesome Cargo Distributor*

2.8.2 UVa 10901 – Ferry Loading III

O problema *Ferry Loading III* consiste no cálculo de tempo que carros levam para cruzar um rio de uma margem a outra utilizando uma balsa. A balsa em questão pode carregar n carros por vez e leva t minutos para cruzar o rio na ida t minutos para cruzar o rio na volta.

Os carros que chegarem tanto pela margem direita quanto pela margem esquerda do rio podem embarcar na balsa se a mesma estiver na mesma margem que o carro chegou e se ainda há vaga na balsa para o embarque, caso a balsa não esteja na margem em que o carro se encontra ou não há mais vagas na balsa o carro deve esperar sua vez.

Analisando o Código Fonte 12 temos a implementação para o problema *Ferry Loading III*. Nela foi implementada contantes *LEFT* e *RIGHT* para indicar a origem em que a balsa dará partida e para controlar o embraque dos carros em cada margem do rio foram implementadas as filas *left* e *right*. Para calcular o tempo total para todos os carro cruzar o rio foi colocado um *loop* que será encerrado apenas quando não houver mais nenhum carro em nenhuma das duas margem do rio. Como a balsa inicialmente inicia seu trabalha a partir da margem da esquerda é por essa fila de carros que começa o embarque e a balsa fica esperando os carros embarcarem até sua lotação máxima e a partir disso ela parte em direção à margem oposta.

Para termos uma implementação única e que funcione para ambos as direções da balsa foi implementado o método *changeDirectionFerry(String current)* quem independente margem atual da balsa o método retorna a margem oposta e dessa forma podemos realizar embarque dos carros independente de que margem eles estejam.

Para cada carro que embarca na balsa é registrado o tempo em que o mesmo chegou na margem somado com o tempo em que a balsa leva para cruzar o rio, esse tempo é armazenado em um vetor para que possamos imprimir cada tempo individualmente ao final de cada caso de teste.

Após os carros embarcarem na balsa é realizado uma verificação, e essa verificação é para ver se ainda possui vaga na balsa para o embarque ou se tempo de chegada do primeiro carro que está esperando o embarque na margem oposta é menor que o tempo corrido no problema(*linha 45 da segunda página de código*) se sim o tempo corrido é atualizado com a soma com o tempo de cruzamento da balsa e a direção da balsa é invertida, caso não seja esse caso na *linha 49 da segunda página* verificamos se os a fila de embarque está vazia ou se o tempo de chegada do primeiro carro da margem oposta é menor que o tempo de chegada do próximo carro da fila da margem atual, se sim o tempo corrido é atualizado com o tempo de chegada do próximo carro da fila da margem oposta somado com o tempo de cruzamento da balsa. Caso nenhum dos dois casos seja verdadeiro o tempo corrido é atualizado com o tempo de chegada do próximo carro da margem atual(*linha 3 da terceira página*).

#	Problem	Verdict	Language	Run Time	Submission Date
24199964	10901 Ferry Loading III	Accepted	JAVA	0.360	2019-11-15 17:37:29

Figura 13: Mensagem de Aceitação do *UVa 10901 – Ferry Loading III*.

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class Main {

    private final Scanner scanner = new Scanner(System.in);
    private final String LEFT = "left";
    private final String RIGHT = "right";

    public static void main(String[] args) {
        Main main = new Main();
        main.run();
    }

    public void run() {
        int numberCases = Integer.parseInt(scanner.nextLine());

        for (int i = 0; i < numberCases; i++) {
            String[] caseConfig = scanner.nextLine().split(" ");
            int maxCountCars = Integer.parseInt(caseConfig[0]);
            int crossingTime = Integer.parseInt(caseConfig[1]);
            int numberCars = Integer.parseInt(caseConfig[2]);

            FerryLoading ferry = new FerryLoading(maxCountCars, crossingTime,
            ↪  numberCars);
            ferry.solve();

            if (i + 1 < numberCases) {
                System.out.println("");
            }
        }
    }

    private class FerryLoading {
        private final Queue<int[]> right;
        private final Queue<int[]> left;
        private final int maxCountCars;
        private final int crossingTime;
        private final int numberCars;

        FerryLoading(int maxCountCars, int crossingTime, int numberCars) {
            this.right = new LinkedList<>();
            this.left = new LinkedList<>();
            this.maxCountCars = maxCountCars;
            this.crossingTime = crossingTime;
            this.numberCars = numberCars;
        }

        public void solve() {
            fillQueues();
```

```
1       int[] crossingTotalTime = calcTime();
2
3       for (int time : crossingTotalTime) {
4           System.out.println(time);
5       }
6   }
7
8   public void fillQueues() {
9       for (int i = 0; i < numberCars; i++) {
10          String[] bufferCar = scanner.nextLine().split(" ");
11          int[] car = {i, Integer.parseInt(bufferCar[0])};
12
13          if (bufferCar[1].equals(LEFT)) {
14              left.add(car);
15          } else {
16              right.add(car);
17          }
18      }
19  }
20
21  public int[] calcTime() {
22      String currentBank = LEFT;
23      Queue<int[]> auxQueue1, auxQueue2;
24
25      int[] finalTimeCars = new int[numberCars];
26      int time = 0;
27
28      while (!left.isEmpty() || !right.isEmpty()) {
29          int countEnqueuedCars = 0;
30
31          if (currentBank.equals(LEFT)) {
32              auxQueue1 = left;
33              auxQueue2 = right;
34          } else {
35              auxQueue1 = right;
36              auxQueue2 = left;
37          }
38
39          while (countEnqueuedCars < maxCountCars && !auxQueue1.isEmpty() &&
            ↪  auxQueue1.peek()[1] <= time) {
40              int[] car = auxQueue1.remove();
41              finalTimeCars[car[0]] = time + crossingTime;
42              ++countEnqueuedCars;
43          }
44
45          if (countEnqueuedCars != 0 || (!auxQueue2.isEmpty() &&
            ↪  auxQueue2.peek()[1] <= time)) {
46              time += crossingTime;
47              currentBank = changeDirectionFerry(currentBank);
48          } else
49          if (auxQueue1.isEmpty() || (!auxQueue2.isEmpty() && auxQueue2.peek()[1]
            ↪  < auxQueue1.peek()[1])) {
50              time = auxQueue2.peek()[1] + crossingTime;
```

```
                    currentBank = changeDirectionFerry(currentBank);
                } else {
                    time = auxQueue1.peek()[1];
                }
            }
            return finalTimeCars;
        }

        public String changeDirectionFerry(String current) {
            return current.equals(LEFT) ? RIGHT : LEFT;
        }
    }
}
```

Código Fonte 12: Solução para o problema *Ferry Loading III*

2.9 Árvore Binária de Pesquisa (balanceada)

2.9.1 UVa 00939 – Genes

O problema consiste que para cada pessoa, sabe-se se o gene é dominante, recessivo ou inexistente. Com base nisso dados, os cientistas formulam uma hipótese sobre como a doença é transmitida de pais para filhos.

A ideia é simples: o programa, dadas as relações pai-filho e uma hipótese fixa, calcular para todas as pessoas se elas têm ou não o gene e, no primeiro caso, se é dominante ou não.

Os casos de Dominante são, se um é dominante e outro dominante ou recessivo. Recessivo se os dois forem recessivo, ou um for dominante e outro não existente. E não existente se um for não existente e o outro não existente ou recessivo.

Para a solução fizemos três funções, uma onde a percorre a árvore genética inteira, arvoreGenetica() e vê quem não tem a informação do gene, caso não tenha chama a verifGenenitica(). Esta por sua vez tenta saber qual é o gene da pessoa, para isso se pergunta qual é o gene dos pais desta pessoa, mas caso os pais também não tem este gene, chama a função novamente para o pai/mãe da pessoa, recursivamente, até achar um gene e com isso vai se completando a árvore com os genes em cada uma das pessoas, os dois primeiros IF, e para caso a pessoa não tenha pai ou mãe, que no caso dos testes isso não ocorre.

A main foi feita pensando numa estrutura de dicionário que guardaria chave/valor, sendo a chave o nome da pessoa e o valor uma lista, onde a posição 0 guarda o gene, a 1 e a 2 guardam os pais, e a partir dai os possíveis filhos que possam ter. De acordo com a entrada é feita uma certa interação, se for nome + gene, já é gravado a pessoa, caso ela não exista, e o gene da pessoa no dicionário, se for nome + nome(filho), guarda o nome da pessoa, caso ela não exista, e o nome do filho, caso ela não exista, sendo guardado o nome do filho na lista do pai, e do pai na lista do filho.

Como tem que estar em ordem alfabética no final, e dicionário não tem uma ordem, existe uma lista que guarda os nomes, e no final ela é ordenada para ser mostrada em ordem ao final do programa.

24181299	939 Genes	Accepted	PYTH3	0.010	2019-11-11 17:59:35
24176848	939 Genes	Runtime error	PYTH3	0.000	2019-11-10 21:31:08

Figura 14: Mensagem de Aceitação do *UVa 00939 – Genes*.

```
def arvoreGenetica(mapa):
    tam = 0
    for j in mapa:
        if (mapa[j][0]==""):
            verifGenetica(j,mapa)
    return mapa
```

Código Fonte 13: Função arvoreGenetica()

```
def verifGenetica(pessoa,mapa):
    primeiro = "";
    segundo = "";
    if mapa[pessoa][1] in mapa:
        primeiro = mapa[pessoa][1]
    if mapa[pessoa][2] in mapa:
        segundo = mapa[pessoa][2]
    if(mapa[primeiro][0] == ""):
        verifGenetica(primeiro,mapa)
    if(mapa[segundo][0] == ""):
        verifGenetica(segundo,mapa)
    if (mapa[primeiro][0] == "dominant" and mapa[segundo][0] ==
    ↪  "non-existent"):
        mapa[pessoa][0] = "recessive"
    elif (mapa[primeiro][0] == "non-existent" and mapa[segundo][0] ==
    ↪  "dominant"):
        mapa[pessoa][0] = "recessive"
    elif (mapa[primeiro][0] == "dominant" and mapa[segundo][0] == "recessive"):
        mapa[pessoa][0] = "dominant"
    elif (mapa[primeiro][0] == "recessive" and mapa[segundo][0] == "dominant"):
        mapa[pessoa][0] = "dominant"
    elif (mapa[primeiro][0] == "dominant" and mapa[segundo][0] == "dominant"):
        mapa[pessoa][0] = "dominant"
    elif (mapa[primeiro][0] == "recessive" and mapa[segundo][0] ==
    ↪  "recessive"):
        mapa[pessoa][0] = "recessive"
    else:
        mapa[pessoa][0] = "non-existent"
    return mapa
```

Código Fonte 14: Função verifGenetica()

```
def main():
    mapa = {}
    num = int(input())
    listaOrd = []
    for i in range(num):
        lista = ["","",""]
        lista2 = ["","",""]
        string = input().split()
        primeiro = string[0];
        segundo = string[1];
        if (not primeiro in mapa):
            mapa[primeiro] = lista
            listaOrd.append(primeiro)
        if (segundo=="dominant" or segundo=="recessive" or
        ↪  segundo=="non-existent") :
            mapa[primeiro][0] = segundo
        else:
            mapa[primeiro].append(segundo)
            if (not segundo in mapa):
                mapa[segundo] = lista2;
                listaOrd.append(segundo)
            if (mapa[segundo][1]==""):
                mapa[segundo][1]=primeiro
            else:
                mapa[segundo][2]=primeiro
    mapa = arvoreGenetica(mapa)
    listaOrd.sort()
    for j in range(len(listaOrd)):
        print(listaOrd[j],mapa[listaOrd[j]][0])
    return 0
if __name__ == "__main__":
    main()
```

Código Fonte 15: Solução para o problema Genes

2.10 Conjuntos

2.10.1 UVa 00978 – Lemmings Battle

O problema consiste em uma batalha até a morte de um ou ambos exércitos de lemmings.

Cada lemming tem um poder, um número inteiro, associado a ele. E cada batalha tem um número de campos de batalhas que podem ser usados, cada campo só pode ter apenas um lemming, sendo no primeiro campo o lemming com mais poder, no segundo, o segundo com mais poder e assim por diante.

Quando a batalha no campo termina o lemming com maior poder sairá vitorioso e o outro morrerá, porém o poder dele será diminuído pelo mesmo tanto que o lemming adversário tinha de poder, caso um lemming verde tenha 40 e o outro lemming azul tenha 30, no final o lemming verde ficará com 10. Caso os dois tenham o mesmo poder, ambos morrem.

Ao final os que sobram lutam de novo e de novo até um dos campos ou ambos morrerem, usando a mesma regra já mencionada acima.

A resolução do problema começa criando os multiset que serão usados, o exército azul e verde, e um armazém para guarda os lemming que ganharam as batalhas.

Um loop que roda a quantidade de guerras que terá. Sempre que começa os exércitos são zerados, e após isso um loop para preencher cada exército.

Na batalha foi feito um iterator para ser um lemming de cada exército, e enquanto algum dos exércitos não fossem vazio, a guerra continuaria. Os lemmings mais fotes lutam, o que ganhar é levado ao armazém com seu poder diminuído, e tirado do exército, depois das batalhas, todos os lemmings do armazém voltam ao exército e novas batalhas começam.

Ao final de tudo ver quem ainda tem alguém no exército ou não, e declara o vencedor e seus soldados, ou empate.

Na linha 28 do Código Início, tem um if perguntando se algum exército é vazio, porque no meio da batalha isto pode ocorrer. Quando não tinha este if o código ficava parado e não sei muito bem o porque, antes eu tava usando .erase(*lemmingX), porém quando tinha repetido, ele deletava todos, mas não precisava do if, depois que coloquei .erase(lemmingX) tive que por este if para não deixar continuar este for se um dos dois forem vazios.

Figura 15: Mensagem de Aceitação do *UVa 00978 – Lemmings Battle*.

```cpp
#include <cstdio>
#include  <iostream>
#include <set>
using namespace std;

int main(){
    multiset<int, greater<int> > eVerde,eAzul,armVerde,armAzul;
    int batalhas;
    scanf("%d",&batalhas);
    for(int i=0;i<batalhas;i++){
        eVerde.clear();eAzul.clear();
        int CB,EV,EA,valor;
        scanf("%d",&CB);scanf("%d",&EV);scanf("%d",&EA);
        for(int v=0;v<EV;v++){
            scanf("%d",&valor);
            eVerde.insert(valor);
        }
        for(int a=0;a<EA;a++){
            scanf("%d",&valor);
            eAzul.insert(valor);
        }

        multiset<int, greater<int> >::iterator lemmingV,lemmingA;
        while(eVerde.size() && eAzul.size()){
            armVerde.clear(); armAzul.clear();
            for(int b=0;b<CB;b++){
                if(!eAzul.empty() && !eVerde.empty()){
                    lemmingA = eAzul.begin();
                    lemmingV = eVerde.begin();
                    if(*lemmingA>*lemmingV){
                        armAzul.insert(*lemmingA-*lemmingV);
                    }
                    else{

                        ↪  if(*lemmingA<*lemmingV){armVerde.insert(*lemmingV-*lemmingA);}
                    }
                    eVerde.erase(lemmingV); eAzul.erase(lemmingA);
                }
                else{break;}
            }
            for (lemmingV = armVerde.begin(); lemmingV != armVerde.end();
            ↪  ++lemmingV){
                eVerde.insert(*lemmingV);
            }
            for (lemmingA = armAzul.begin(); lemmingA != armAzul.end();
            ↪  ++lemmingA){
                eAzul.insert(*lemmingA);
            }
        }
```

Código Fonte 16: Solução para o problema dos Lemmings Battle - Início

```
        if(eVerde.empty() && eAzul.empty()){
            printf("green and blue died\n");
        }
        else{
            if(eVerde.empty()){
                printf("blue wins\n");
                for (lemmingA = eAzul.begin(); lemmingA != eAzul.end();
                ↪  ++lemmingA){
                    cout << *lemmingA << "\n";
                }
            }
            else{
                printf("green wins\n");
                for (lemmingV = eVerde.begin(); lemmingV != eVerde.end();
                ↪  ++lemmingV){
                    cout << *lemmingV << "\n";
                }
            }
        }
        if(i<batalhas-1){
            printf("\n");
        }
    }

};
```

Código Fonte 17: Solução para o problema dos Lemmings Battle - Final

2.10.2 UVa 11849 – CD

O problema do CD consiste em verificar quais cds jack e jill tem em comum para vender enquanto eles valem algo.

Analisando Código Fonte 18, o código referente das linhas 7 a 9 é responsável por tratar a primeira entrada, onde 'n' e 'm' é respectivamente os valores dos CDS do Jack e do Jill,sendo esses valores diferente de 0. Após essa leitura, é adicionado em um *HashSet* os n cds do Jack (linha 14 a 17) e em seguida é adicionado os cds do Jill, verificando a cada iteração se já tem o cd, caso ele esteja, é incrementado a variável *acc* responsável por computar os cds repetidos que serão vendidos, caso não esteja na estrutura, é adicionado(linha 19 a 29).

Para solucionar esse problema, inicialmente foi desenvolvido com o auxilio de uma *TreeSet* que se baseia na arvore vermelho-preto, nessa estrutura, não importa a ordem que for adicionado um elemento, eles serão ordenados, por isso o resultado que obtivemos ao submeter no judge online foi *Time limit exceeded*, para reverter o quadro de TLE, alteramos a estrutura para *HashSet* que usa um *Hash Table* e com isso, o tempo não passou do limite.

#	Problem	Verdict	Language	Run Time	Submission Date
24163779	11849 CD	Accepted	JAVA	2.970	2019-11-07 19:20:51
24163328	11849 CD	Time limit exceeded	JAVA	3.000	2019-11-07 17:09:35

Figura 16: Mensagem de Aceitação do *UVa 11849 – CD*.

```java
import java.util.HashSet;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        int m= scan.nextInt();
        int acc =0;
        while(n!=0 && m != 0){
            //cria HashSet
            HashSet<Integer> tsCds = new HashSet<Integer>();
            //Adiciona os cds do Jack na Árvore
            for(int i=0;i<n;i++){
                int cdsJack = scan.nextInt();
                tsCds.add(cdsJack);
            }
            //Adiciona os cds do Jill
            for(int i=0;i<m;i++){
                int cdsJills = scan.nextInt();
                //Caso o CD esteja na estrutura, soma no acumulador
                //Caso não, adiciona na estrutura.
                if(tsCds.contains(cdsJills)){
                    acc++;
                }else{
                    tsCds.add(cdsJills);
                }
            }
            System.out.println(acc);
            acc =0;
            n = scan.nextInt();
            m= scan.nextInt();
        }
    }
}
```

Código Fonte 18: Solução para o problema do CD