Universidade Federal do Vale do São Francisco – UNIVASF

Breno Gabriel de Souza Coelho

**Explanação acerca do processo de**

**Resolução das questões selecionadas**

Petrolina – PE

2020

Metodologia

Pelo exposto em sala e na descrição do trabalho, eu entendo que o objetivo desse trabalho é analisar a compressão do aluno sobre os tópicos estudados na disciplina e sua capacidade de utilizar esses conhecimentos como ferramentas na resolução de problemas.

Eu decidi construir o trabalho que eu gostaria de receber: aqui eu descrevo em detalhes não só o funcionamento do código, mas a lógica por detrás dele e o pensamento que me fez chegar a ele.

A ideia é dividir cada explicação em etapas; etapas essas que são seguidas por mim mesmo quando penso em resolver um problema qualquer de computação. Primeiramente eu busco um algoritmo/sequência de passos que eu, como humano, posso utilizar para chegar ao resultado. Ao mesmo tempo penso também em como poderia, através da linguagem de programação, “ordenar” o computador para que ele realizasse esses passos. Uma vez que tenho uma visão clara de como e onde pretendo chegar, construo o código fonte, onde dou vida aos algoritmos pensados.

A divisão de cada resolução terá 7 tópicos: 3 deles voltados para explicação em sí, e 4 voltados para análise e exposição de coisas pertinentes.

Eu optei por colocar em cada resolução o enunciado de cada problema (para tornar mais prático o acesso a ele), o código-fonte que eu construí para responder a questão (que também acompanha esse arquivo, porém mais uma vez é prático tê-lo para acompanhar a descrição com mais facilidade).

O arquivo ficou um pouco extenso, mas o encorajo a lê-lo inteiro, não irá se arrepender! Esse tamanho também se deve ao fato de que eu coloco várias partes do código fonte no decorrer da explicação, além do enunciado e do código-fonte inteiro.

Cada resolução é dividida em 7 partes:

**1) Descrição do Problema:** Onde eu apresento a descrição presente no URI.

**2) Algoritmo de Resolução:** Onde descrevo o algoritmo o qual eu quero que o computador realize.

**3) Pensando no algoritmo em termos computacionais:** Nessa parte eu descrevo como algumas ideias centrais do algoritmo poderiam ser feitas código.

**4) Código-Fonte:** O código-fonte da resolução.

**5) Explicação do código-fonte:** A explicação em detalhes de cada parte do código-fonte.

**6) Observações finais:** Eventuais observações que eu achei pertinentes (algumas questão não tem nenhuma)

**7) Tópicos abordados:** Uma seleção dos principais tópicos (a meu ver) da disciplina que foram abordados na questão.

Algumas explicações terão o tópico 2 e 3 bem reduzidos, porque sua lógica de resolução é simples.

Segue abaixo a minha lista pessoal dos principais tópicos vistos na disciplina:

**1)Fundamentos de C:** Estrutura geral, Diretivas de Pré-processamento e Função main.

**2)Fundamentos de C:** Dados primitivos, operadores matemáticos e relacionais, funções de entrada e saída de dados (printf e scanf)

**3)Fundamentos de C: Saída** formatada de dados.

**4)Estruturas de Controle de Fluxo:** Desvios (simples, composto, aninhado, encadeado, switch)

**5)Estruturas de Controle de Fluxo:** Laços (for, while e Do while)

**6)Estruturas de Controle de Fluxo:** Comandos Continue, Break e Exit;

**7)Vetores**: Declaração, Inicialização e Manipulação

**8)Strings:** Declaração, Inicialização e Manipulação

**9)Matrizes:** Declaração, Inicialização e Manipulação.

**10)Funções de manipulação de Strings:** gets, puts, strlen, strcmp, strcat, strcpy, strstr, strchr.

**11)Ponteiros:** Definição, Declaração, Inicialização, Vetores como ponteiros, Ponteiros de Ponteiros.

**12)Funções:** Estrutura, Entrada e saida de dados, passagem de parâmetros por valor e por referência, protótipos de funções.

**13)Escopo de Variáveis:** Globais, Locais, Formais.

**14)Estruturas de Dado Heterogêneas (Struct):** Declaração, Inicialização, Manipulação (operador ‘.’), Ponteiros para Struct (operador →)

**15)Alocamento Dinâmico de Memória:** Funções malloc, calloc e realloc.

**16) Arquivos**

(Os tópicos teóricos, abordados antes da aula 5, não foram mencionados pois eles não são usados diretamente no código – mas eventualmente posso comentar algo envolvendo conhecimentos obtidos ao estudá-los).

O tópico de arquivos não será abordado em questão alguma, pois o URI lê apenas o código fonte.

**- Sobre a estrutura do trabalho:**

Apenas optei por uma forma organizada e clara, em alguns aspectos inspirada pelas normas ABNT (como a capa). A fonte é Arial tamanho 12 para o texto em geral, tópicos serão destacados em negrito e a linguagem é informal, em geral eu vou prezar pela objetividade e clareza, mas as vezes posso colocar um pouco de emoção no texto kkk.

Esse arquivo foi construído usando o programa LibreOffice Writer (que é gratuito); no entanto esse gera, nativamente, programas com extensão **“.odt”** . A versão **“.word”** que acompanha a **“.odt”** foi gerada a partir de uma conversão que o próprio programa realiza entre esses textos; no entanto ele alerta para a possibilidade de alguns problemas de formação.

Assim algum problema dessa natureza deve ser oriundo exatamente da conversão. Caso o professor tenha o Libre Office Writter instalado, talvez seja melhor ver o arquivo **“.odt”** (que é a formatação original). Mas creio que nenhum problema severo deve ocorrer.

1766

O Elfo das Trevas

**1) Descrição do Problema:**

(Presente em https://www.urionlinejudge.com.br/judge/pt/problems/view/1766)

O estábulo onde ficam as renas foi intencionalmente aberto pelo Elfo das Trevas permitindo que cada uma delas corresse e voasse livremente pela fábrica do Papai Noel, causando o maior transtorno. Os elfos estão tentando desesperadamente fazer o possível para deixar o trenó pronto para embarque. Você ficou responsável por colocar cada rena na sua posição correta assim que ela é capturada por um dos outros elfos.

Você sabe que o estábulo segue uma organização baseada na ordem que as renas irão ocupar no trenó. Desta forma, na hora da partida todas podem ser facilmente posicionadas. Diferentemente do que muitos pensam, as renas são posicionadas em uma fila única à frente no trenó. Nem todas as renas do estábulo são utilizadas em cada viagem, isto depende da carga total do trenó.

Você conseguiu a lista com as características que são utilizadas para determinar a ordem de rena. Elas devem ser ordenadas primeiramente de forma decrescente por peso. Caso duas ou mais apresentarem o mesmo peso elas devem ser ordenadas de forma ascendente pela idade, após pela altura e caso ainda persista empate, pelo nome.

Utilizando seu computador mágico de última geração você quer escrever um programa que ordene as renas, de acordo com as características informadas, e exiba somente o número exato de renas que serão utilizadas no trenó (de forma ordenada).

**Entrada:**

Este problema possui diversos casos de teste. A primeira linha de entrada contém um inteiro **T** (1 ≤ **T** ≤ 10^5) que indica o número de casos de teste a seguir. A primeira linha de cada caso de teste contém dois inteiros **N** e **M** (5 ≤ **N, M** ≤ 10^3) que indicam respectivamente o número total de renas e o número de renas que irão puxar o trenó. Na sequência serão informados uma string S seguida por 2 inteiros P (1 ≤ P ≤ 300) e I (1 ≤ I ≤ 300) e por um número de ponto flutuante A (0.00 ≤ A ≤ 3.00), indicando respectivamente o nome, o peso, a idade e a altura de cada uma das renas. O nome de cada uma das renas é composto somente por uma palavra com até 100 caracteres.

**Saída:**

Para cada caso de teste você deverá exibir a mensagem "CENARIO {i}", onde i indica o caso de teste atual, seguido da posição e o nome de cada umas das M renas que irão puxar o trenó, ordenadas conforme descrito acima.

|  |  |
| --- | --- |
| Exemplo de Entrada | Exemplo de Saída |
| 1 9 5 Rudolph 50 100 1.12 Dasher 10 121 1.98 Dancer 10 131 1.14 Prancer 7 142 1.36 Vixen 50 110 1.42 Comet 50 121 1.21 Cupid 50 107 1.45 Donner 30 106 1.23 Blitzen 50 180 1.84 | CENARIO {1} 1 - Rudolph 2 - Cupid 3 - Vixen 4 - Comet 5 - Blitzen |

**2)Algoritmo de Resolução:**

Pela descrição do problema já fica claro as instruções que o programa (ou um ser humano) deveria seguir para resolvê-lo:

- Organizar as renas da mais pesada para a mais leve

- Se duas tiverem o mesmo peso, fica na frente a que for mais nova (menor idade)

- Se as idades forem iguais também, fica na frente a mais baixa (menor altura);

- Se as alturas também forem iguais, elas serão organizadas em ordem alfabética pelo nome

Bom, se os nomes também forem iguais então as renas basicamente são iguais, então tanto faz kkk. Na realidade acho que não existe nenhum caso teste onde isso ocorre, mas o algoritmo funcionaria mesmo se esse fosse um caso.

**3) Pensando no Algoritmo em Termos Computacionais:**

Certo, agora basta encontrar uma forma de construir um programa, em C, que leia os dados de entrada, realize o algoritmo de resolução e então apresente as saídas. Como cada rena é caracterizada por uma série de tipos de dados diferentes (uma string para o nome, dois inteiros para o peso e idade e um float para altura), parece interessante construir uma Estrutura que incorpore todos esses dados de uma vez, como um dado “Rena”. Isso é bem simples de fazer:

typedef Struct{

char nome[101];

int Peso;

int Idade;

float Altura;

} Rena;

Assim eu crio um novo tipo de dado nomeado de “Rena” e que possui exatamente as informações que caracterizam cada uma delas nesse contexto. Agora basta ler todas as **R** *renas,* cada qual sendo um estrutura de nome rena, o que torna útil criar um vetor de renas.

Uma vez que eu tenha um vetor com todas as R renas, basta passá-las pelo algoritmo de Resolução, que poderia ser construído comparando os campos apropriados de cada rena(pesos, idades, etc) e utilizando um algoritmo de organização simples. Por exemplo, se eu tenho um vetor {9, 4, 5, 5, 12} eu posso ordená-lo do maior para o menor usando o seguinte algoritmo de organização:

-Começo com o segundo elemento (4)

-Comparo todos os elementos anteriores a ele com ele, se ele for maior eu troco os dois de posição, fazendo com que o maior fique na frente (se não for maior, nada faço)

-Continuo fazendo isso até terminem os elementos antes dele

-Agora passo para o próximo elemento (terceiro) e repito o algoritmo a partir do segundo passo.

Aplicando isso no vetor dado, teríamos:

{9, 4, 5, 5, 12} (começo com 4, assim comparo 9 (primeiro elemento) e 4, como 4 não é maior que 9, nada faço)

{9, 4, 5, 5, 12} (como 9 era o único elemento antes de 4, passo agora para 5. Comparando 5 e 9 (primeiro elemento), vejo que 5 < 9 portanto nada faço)

{9, 4, 5, 5, 12} (agora comparo 4 (segundo elemento) e 5, como 5 é maior, troco os dois de posição)

{9, 5, 4, 5, 12} (acabaram os elementos antes de 4, então passo para 5 (quarta posição). Comparo 5 e 9 (primeira posição), como 5 < 9, nada faço)

{9, 4, 5, 5, 12} (agora comparo 4 (segunda posição) e 5, como 5 > 4, troco eles de posição)

{9, 5, 5, 4, 12} (agora o elemento da quarta posição é 4, e não mais 5, por conta da troca na etapa anterior, 4 < 5 (terceira posição), então nada faço e passo para o 12)

{9, 4, 5, 5, 12} (12 > 9 (primeira posição), logo troco os dois de posição)

{12, 4, 5, 5, 9} (9 é o novo valor na quinta posição. 9 > 4 (segunda posição), logo troco)

{12, 9, 5, 5, 4} (4 é o novo valor na quinta posição. 4 < 5 (terceira posição) logo nada faço)

{12, 9, 5, 5, 4} (novamente 4 < 5 (quarta posição), então nada faço)

O algoritmo chega ao final e de fato ordenamos o vetor dado. Usando esse método podemos também ordenar em ordem decrescente, e ele será a base lógica para ordenação das renas por peso, idade e altura.

(Algum tempo depois de escrever isso descobri que esse algoritmo, que eu havia “descoberto” enquanto fazia questões para ordenar vetores, é uma variação de um algoritmo chamado *Insertion Short*)

Uma vez que eu ordene todas as renas, posso exibir na tela, através de um printf, apenas o nome das **P** que irão *puxar* o trenó, usando a formatação de saída adequada. Esse foi basicamente meu ponto de partida para construção do código-fonte.

**4) Código-Fonte:**

**#include <stdio.h>**

**#include <stdlib.h>**

**typedef struct{**

**char nome[101];**

**int Peso;**

**int Idade;**

**float Altura;**

**} Rena;**

**Rena \*pRena;**

**void Troca (int i, int j)**

**{**

**Rena aux;**

**aux = \*(pRena + j);**

**\*(pRena + j) = \*(pRena + i);**

**\*(pRena + i) = aux;**

**return;**

**}**

**int ComparaNome(char \*vet1, char \*vet2)**

**{**

**int i;**

**for (i = 0; \*(vet1 + i) != '\0'; i++){**

**if (vet1[i] > vet2[i]) return 1;**

**if (vet1[i] < vet2[i]) return 0;**

**}**

**}**

**main ()**

**{**

**long int N;**

**int R, P, i, j, cont;**

**scanf ("%ld", &N);**

**for (cont = 0; cont < N; cont++){**

**scanf ("%d %d", &R, &P);**

**/\* Construindo um vetor com R renas usando alocação dinamica \*/**

**pRena = (Rena\*) malloc(R \* sizeof(Rena));**

**/\*Lendo os dados de cada rena \*/**

**for (i = 0; i < R; i++){**

**scanf ("%s", (pRena + i) -> nome);**

**scanf ("%d", &((pRena + i) -> Peso));**

**scanf ("%d", &((pRena + i) -> Idade));**

**scanf ("%f", &((pRena + i) -> Altura));**

**}**

**/\* Organizando o vetor de renas segundo a ordem especificada\*/**

**for (i = 1; i < R; i++){**

**for (j = 0; j < i; j++){**

**if ((pRena + j) -> Peso < (pRena + i) -> Peso)**

**Troca(i, j);**

**else if ((pRena + j) -> Peso == (pRena + i) -> Peso){**

**if ((pRena + j) -> Idade > (pRena + i) -> Idade)**

**Troca(i, j);**

**else if ((pRena + j) -> Idade == (pRena + i) -> Idade){**

**if ((pRena + j) -> Altura > (pRena + i) -> Altura)**

**Troca(i, j);**

**else if ((pRena + j) -> Altura == (pRena + i) -> Altura)**

**if (ComparaNome((pRena + j) -> nome, (pRena + i) -> nome))**

**Troca(i, j);**

**}**

**}**

**}**

**}**

**printf ("CENARIO {%d}\n", cont + 1);**

**for (i = 0; i < P; i++)**

**printf ("%d - %s\n", i + 1, (pRena + i) -> nome);**

**}**

**free(pRena);**

**}**

**5) Explicação do código-fonte**

Minha ideia é construir um vetor de renas, e optei por usar uma alocação dinâmica na criação deste, em parte como um desafio para mim mesmo (já que usar um vetor normal de renas parecia muito com outras coisas que já fiz antes), e em parte para expandir o leque de conteúdos abordados.

**#include <stdio.h>**

**#include <stdlib.h>**

Inicio meu código adicionando as diretivas de pré-processamento **stdio.h** (que será necessária para ultilizar as funções de entrada e saída – printf e scanf - ) e **stdlib.h** (que será necessária para usar alocação dinâmica através da função malloc).

Meu codigo-fonte acabou contando com duas funções que manipulam o vetor de renas, então para simplificar as coisas vou definir o dado tipo rena e o ponteiro para o vetor de renas globalmente:

**typedef struct{**

**char nome[101];**

**int Peso;**

**int Idade;**

**float Altura;**

**} Rena;**

**Rena \*pRena;**

Agora vamos analisar a função main, começando pela parte mais “externa”:

**main ()**

**{**

**long int N;**

**int R, P, i, j, cont;**

**scanf ("%ld", &N);**

**for (cont = 0; cont < N; cont++){**

**…**

**}**

Início criando a função main normalmente e definindo as variáveis que usarei no decorrer do código.

Já possuo definido o ponteiro **pRena,** que será o início do meu vetor de renas. Defino também três contadores: **i, j**  e **cont; i** e **j** serão usados para o algoritmo de organização, enquanto **cont** serve para contar os **N** casos de entrada.

**R** é o número total de renas, **P** é o número das que puxarão o trenó.

**N** foi definido como long int pois na questão é informado que 1 ≤ **T** ≤ 10^5 (T é o número de casos testes, no texto do problema), e um inteiro simples vai de -32.767 até 32.767, e um unsigned int vai de 0 a 65.535, como informado na página 153 do material da disciplina (ou slide 3 da aula 07), valores insuficientes para abordar o limite de casos de teste. Long int, com base no mesmo slide, passa tranquilamente esse valor, o que justifica minha escolha.

Em seguida é lido o número **N** de casos de teste a seguir através do scanf, e o ciclo for, que opera com o contador **cont**, é responsável por rodar o código para cada caso de teste.

**/\* Construindo um vetor com R renas usando alocação dinamica \*/**

**pRena = (Rena\*) malloc(R \* sizeof(Rena));**

**/\*Lendo os dados de cada rena \*/**

**for (i = 0; i < R; i++){**

**scanf ("%s", (pRena + i) -> nome);**

**scanf ("%d", &((pRena + i) -> Peso));**

**scanf ("%d", &((pRena + i) -> Idade));**

**scanf ("%f", &((pRena + i) -> Altura));**

**}**

Como os comentários que coloquei já descrevem, primeiro eu aloco a memória necessária para ler as **R** renas. A função malloc devolve um ponteiro para o inicio da memória alocada, que é capturado pelo meu ponteiro de renas **pRena.**

Em seguida eu leio os dados das **R** renas, cada entrada, como mostrado na descrição do problema, conterá o nome da rena, seu peso, idade e altura. O ponteiro **pRena** aponta para o início do vetor de renas, então qualquer ponteiro do vetor de renaspode ser escrito como **pRena + i,** onde **0<= i <= R,** em seguida uso o operador “→” para acessar um campo da estrutura apontada por **pRena + i.** Uma forma mais ortodoxa de fazer isso é usando o operador **“.”** e lembrando que \*(pRena + i) é a mesma coisa que pRena[i], dai o conteudo de um campo seria **pRena[i].nomedocampo**.

***OBS:******(pRena + i) → Peso*** *se refere ao conteúdo do campo Peso da estrutura apontada por (pRena + i), então preciso colocar um “&” na frente de* ***(pRena + i) → Peso*** *ao ultiliza-la no scanf, já que o scanf exige um ponteiro. Como o conteúdo de* ***(pRena + i) → nome*** *é um ponteiro para o início do vetor nome[101], esse “&” se torna desnecessário (para esse caso).*

Feitos esses esclarecimentos, os scanf’s leem as entradas e as colocam no campo apropriado (string vai pra (pRena + i) → nome, peso vai pra (pRena + i) → Peso e etc), o ciclo **for** garante que serão lidas as informações de todas as R renas e essas serão colocadas numa posição própria em cada vetor; assim esse vetor se torna uma descrição precisa de R renas.

***OBS:*** *Não é necessário limpar o buffer, pois %s não captura espaços nem “\n”.*

Na parte seguinte do código estará presente a função **Troca (i, j)**, que basicamente troca as Structs na posição **i** e **j** de posição no vetor de structs. Vamos começar falando sobre como essa função funciona.

**void Troca (int i, int j)**

**{**

**Rena aux;**

**aux = \*(pRena + j);**

**\*(pRena + j) = \*(pRena + i);**

**\*(pRena + i) = aux;**

**return;**

**}**

Ela recebe **i e j,** posições das structs no vetor e que pretendemos trocar. Para isso é utilizado um auxiliar **aux** do tipo rena, esse auxiliar recebe o conteúdo de **pRena + j**, assim “guardamos” o conteúdo desse para uso futuro, e já que ele foi guardado podemos sobrescrever o conteúdo na posição onde ele estava com o conteúdo de **pRena + i**. Agora **pRena + i** e **pRena + j** possuem o mesmo conteúdo.

Para finalizar queremos que **pRena + i** receba o antigo conteúdo de **pRena + j**, já salvo em **aux**. Dai basta fazer **pRena + i** receber o conteudo de **aux**, finalizando a troca.

Como **pRena + i** e **pRena + j** são ponteiros definidos globalmente, nenhum retorno é necessario.

Tendo isso em mente, vamos voltar para a função main:

**/\* Organizando o vetor de renas segundo a ordem especificada\*/**

**for (i = 1; i < R; i++){**

**for (j = 0; j < i; j++){**

**if ((pRena + j) -> Peso < (pRena + i) -> Peso)**

**Troca(i, j);**

**else if ((pRena + j) -> Peso == (pRena + i) -> Peso){**

**if ((pRena + j) -> Idade > (pRena + i) -> Idade)**

**Troca(i, j);**

**else if ((pRena + j) -> Idade == (pRena + i) -> Idade){**

**if ((pRena + j) -> Altura > (pRena + i) -> Altura)**

**Troca(i, j);**

**else if ((pRena + j) -> Altura == (pRena + i) -> Altura)**

**if (ComparaNome((pRena + j) -> nome, (pRena + i) -> nome))**

**Troca(i, j);**

**}**

**}**

**}**

}

O primeiro ciclo **for** é responsável por mover o contador **i** da posição 1 até o final **R** do vetor de renas, e o segundo **for** moverá o contador **j** da posição inicial 0 até a posição **i.** Observe portanto que **j < i** sempre.

A ciclo **j** haverá, primeiramente, uma comparação entre o peso da rena na posição **(pRena + j)** e aquela na posição **(pRena + i)**, se o peso daquela for menor que o dessa então elas devem ser trocadas, pois as mais pesadas devem ficar na frente e j < i , e tal troca é efetuada pela função Troca( i, j ).

Essa comparação, assim como as que virão a frente, é feita comparando os campos apropriados das renas. Sendo assim, para comparar os pesos das renas mencionadas, comparamos os valores de **(pRena + j) → Peso** e **(pRena + i) → Peso.**

Para comparar as idades, **(pRena + j) → Idade** e **(pRena + i) → Idade.**

Caso o peso daquela seja maior ou igual ao dessa entramos no caso **else**. Imediatamente é verificado, por um **if,** se o peso das renas é igual; se não for, então nada deve ser feito (pois o peso da mais a frente já é maior que o daquela mais atrás, o que atende ao especificado), já se o peso de fato for igual entramos em um novo caso **if.**

Nesse é primeiramente verificado se a idade da rena mais a frente (j) é maior que a daquela mais atrás (i), se sim elas são trocadas pela função antes mencionada (pois as mais novas devem ficar mais a frente).

Contudo, se o caso não for esse, o programa acessa mais um **else.** É imediatamente verificado, por um **if,** se a idade delas é igual; se não for, então nada deve ser feito (pois a mais nova já está a frente), mas se for então entramos em mais um **if.**

É analisada a relação entre suas alturas, se a mais a frente for mais alta que a mais atrás então elas são trocadas. Se não, o caso **eles** é ativado e é imediatamente verificado se as alturas são iguais, se não forem então nenhuma mudança se faz necessária, se forem então o programa deve organizá-las por ordem alfabética.

Para fazer isso é utilizada a função ComparaNome:

**int ComparaNome(char \*vet1, char \*vet2)**

**{**

**int i;**

**for (i = 0; \*(vet1 + i) != '\0'; i++){**

**if (vet1[i] > vet2[i]) return 1;**

**if (vet1[i] < vet2[i]) return 0;**

**}**

}

Ela recebe o ponteiro para o início da string nome da primeira e segunda struct (**ComparaNome((pRena + j) -> nome, (pRena + i) -> nome )** como sendo vetores, e então compara posições iguais.

Todo caractere é interpretado como um número do código ASCII pelo compilador, então as comparações **vet1[i] > vet2[i]** e **vet1[i] < vet2[i]** irão, na verdade, comparar os valores da tabela ASCII correspondentes a esses caracteres. Como o alfabeto (tanto o maiúsculo quanto minusculo) está organizado de maneira crescente no código ASCII, números menores se referem a letras que vem primeiro no alfabeto.

Se as letras comparadas forem iguais, nada ocorre e o ciclo simplesmente avança. Quando elas forem diferentes um dos **if**’s será ativado, caso a letra do nome da rena mais a frente (j) vier depois (ou seja, seu número no ASCII for maior) que a letra do nome da rena mais atrás (i), a função retorna o valor 1, que fará a função troca ser ativada.

Já se a ordem estiver correta é enviado o valor 0, que faz com que nada aconteça (pois as renas já estão da maneira certa).

Esse processo de organização será repetido várias vezes, e como pode-se perceber ele é a forma, em código, do algoritmo de organização antes mencionado. Ao fim as **R** renas estão todas ordenadas, contudo só me é exigido mostrar as **P** primeiras:

**printf ("CENARIO {%d}\n", cont + 1);**

**for (i = 0; i < P; i++)**

**printf ("%d - %s\n", i + 1, (pRena + i) -> nome);**

Aqui eu apresento, com a formatação exigida, todas as **P** primeiras renas do conjunto de **R** renas já organizadas através da utilização do ciclo **for.**

Note que foi exigida a apresentação de “CENARIO { }”, onde o número dentro das chaves é o número do caso de teste em ordem comum (isto é, sem o 0. Portanto 1, 2, 3 …), por isso ocorre a adição de 1 no **cont** apresentado (lembre que **cont** é um contador voltado somente para o número do caso de teste). Algo semelhante ocorre ao apresentar o número da posição **i.**

Antes de finalizar o programa, a linha de comando **free(pRena)** libera a memória alocada para o vetor **pRena** através da função **malloc,** evitando um *memoria leak* (que seria bem pequeno, mas é bom criar o costume de liberar a memória alocada dinamicamente).

**6) Observações finais:**

-A função **ComparaNome** poderia receber somente **i** e **j**, e então utilizar diretamente as strings (pRena + i) → nome e (pRena + j) → nome, já que pRena é um ponteiro global. Mas eu optei por recebê-las como vetores, pois assim eu faço uma coisa diferente da que fiz em troca (utilizando outro conhecimento), além da função construída servir também para outros programas (onde os vetores que pretendo organizar não necessariamente vão ser globais).

**7) Tópicos abordados na questão:**

Com base na minha lista pessoal, todos os seguintes tópicos estão presentes no código:

**1)Fundamentos de C:** Estrutura geral, Diretivas de Pré-processamento e Função main.

**2)Fundamentos de C:** Dados primitivos, operadores matemáticos e relacionais, funções de entrada e saída de dados (printf e scanf)

**4)Estruturas de Controle de Fluxo:** Desvios (simples, composto, encadeado)

**5)Estruturas de Controle de Fluxo:** Laços (for)

**8)Strings:** Declaração, Inicialização e Manipulação

**10)Ponteiros:** Definição, Declaração, Inicialização, Vetores como ponteiros.

**11)Funções:** Estrutura, Entrada e saida de dados, passagem de parametros por valor e por referência.

**12)Escopo de Variaveis:** Globais, Locais, Formais.

**13)Estruturas de Dado Heterogeneas (Struct):** Declaração, Inicialização, Ponteiros para Struct (operador →).

**14)Alocamento Dinâmico de Memória:** Funções malloc.

1120

Revisão de Contrato

**1) Descrição do Problema:**

(Presente em <https://www.urionlinejudge.com.br/judge/pt/problems/view/1120>)

Durante anos, todos os contratos da Associação de Contratos da Modernolândia (ACM) foram datilografados em uma velha máquina de datilografia.  
  
Recentemente Sr. Miranda, um dos contadores da ACM, percebeu que a máquina apresentava falha em um, e apenas um, dos dígitos numéricos. Mais especificamente, o dígito falho, quando datilografado, não é impresso na folha, como se a tecla correspondente não tivesse sido pressionada. Ele percebeu que isso poderia ter alterado os valores numéricos representados nos contratos e, preocupado com a contabilidade, quer saber, a partir dos valores originais negociados nos contratos, que ele mantinha em anotações manuscritas, quais os valores de fato representados nos contratos. Por exemplo, se a máquina apresenta falha no dígito 5, o valor 1500 seria datilografado no contrato como 100, pois o 5 não seria impresso. Note que o Sr. Miranda quer saber o valor numérico representado no contrato, ou seja, nessa mesma máquina, o número 5000 corresponde ao valor numérico 0, e não 000 (como ele de fato aparece impresso).

**Entrada**

A entrada consiste de diversos casos de teste, cada um em uma linha. Cada linha contém dois inteiros **D** e **N** (1 ≤ **D** ≤ 9, 1 ≤ **N**< 10^100 ), representando, respectivamente, o dígito que está apresentando problema na máquina e o número que foi negociado originalmente no contrato (que podem ser grande, pois Modernolândia tem sido acometida por hiperinflação nas últimas décadas).  
  
O último caso de teste é seguido por uma linha que contém apenas dois zeros separados por espaços em branco.

**Saída**

Para cada caso de teste da entrada o seu programa deve imprimir uma linha contendo um único inteiro V, o valor numérico representado de fato no contrato.

| Exemplo de Entrada | Exemplo de Saída |
| --- | --- |
| 5 5000000 3 123456 9 23454324543423 9 99999999991999999 7 777 0 0 | 0 12456 23454324543423 1 0 |

**2)Algoritmo de Resolução:**

Primeiramente observe que os valores do contrato são muito grandes, indo até 10^100, então uma variável inteira para armazenar isso está fora de cogitação. A melhor maneira de lidar com um número tão grande é pensar nele como uma string; assim para eliminar os valores defeituosos basta analisar elemento a elemento dele, aplicando o procedimento abaixo:

-Se for igual ao valor da tecla defeituosa, eu o elimino (troco por um espaço vazio por exemplo).

-Se for diferente, o mantenho.

Após feito isso, é esperado que tenha sobrado uma string com uma série de espaços vazios. Para encontrar o valor desejado, precisamos primeiro “juntar” esses números espaçados, e daí analisar o resultado.

E é exatamente nessa análise que reside um detalhe fundamental, pois essa junção pode ter vários zeros no início, e zeros à esquerda não valem nada. Assim, antes de exibir o resultado final, precisamos eliminar todos os zeros à esquerda antes do primeiro número não nulo.

Por exemplo, se o número for 1000125 e o dígito defeituoso for 1, seguindo esses passos eu teria:

1) { ,0,0,0, ,2,5} (troquei todos os elementos iguais ao dígito defeituoso por um espaço vazio)

2) {0,0,0,2,5} (juntei tudo)

3) {2,5} (eliminei os zeros antes do primeiro dígito não nulo)

(coloquei as chaves para deixar mais claro)

O valor procurado então seria 25.

**3) Pensando no Algoritmo em Termos Computacionais:**

O primeiro passo do algoritmo (procurar elementos iguais ao dígito defeituoso e elimina-los) é fácil de implementar em código, pois se trata de uma busca simples.

A atenção reside então em como juntar as partes após a eliminação. A alternativa que eu decidi seguir foi colocar os elementos diferentes da tecla defeituosa diretamente em uma nova string.

Agora para eliminar os dígitos nulos antes do primeiro não nulo eu poderia fazer uma nova busca, encontrando a posição **i** na string onde ocorre o primeiro dígito não nulo e então mostrar na tela somente a string **(vet + i) !**

**4) Código-Fonte:**

**#include <stdio.h>**

**main ()**

**{**

**int a, aux2, i, cont, r = 1;**

**char vet[101], aux[101];**

**while (r){**

**scanf ("%d %s", &a, vet);**

**if (a == 0 && vet[0] == '0') break;**

**for (i = 0, cont = 0; vet[i] != '\0'; i++){**

**aux2 = vet[i];**

**aux2 -= 48;**

**if (aux2 != a){**

**aux[cont] = vet[i];**

**cont++;**

**}**

**}**

**aux[cont] = '\0';**

**if (cont == 0) printf ("0\n");**

**else{**

**for (i = 0; aux[i] == '0'; i++);**

**if (i < cont) printf ("%s\n", aux + i);**

**else printf ("0\n");**

**}**

**}**

**}**

**5) Explicação do Código-Fonte:**

No início eu adiciono ao programa o arquivo header stdio.h, através da diretiva #include, para poder usar funções como printf, scanf, etc. Construo a função main normalmente, e declaro algumas variáveis:

**int a, aux2, i, cont, r = 1;**

**char vet[101], aux[101];**

A variável **a** receberá o valor do dígito defeituoso, **aux2** é um inteiro auxiliar que será importante numa certa parte do código, **i e cont** são contadores, **r = 1** será usado para manter um ciclo infinito, o valor do contrato será colocado em **vet,** e **aux** receberá tudo em vet que seja diferente do dígito defeituoso.

**#include <stdio.h>**

**main ()**

**{**

**int a, aux2, i, cont, r = 1;**

**char vet[101], aux[101];**

**while (r){**

**(...)**

**}**

**}**

O ciclo while é “infinito”, a priori, pois o valor de **r** se manterá inalterado no código. Porém uma condição interna permitirá a ocorrência de um **break**, finalizando-o.

Coloquei dessa maneira pois se o dígito inserido e o valor forem ambos 0, o programa deve finalizar sua execução. Sendo assim, ou eu ultilizaria um **do while** (mas nesse caso pelo menos um ciclo de processamento ocorreria, e eu queria que caso fosse inserido 0 0 a qualquer momento o programa finalizasse sua execução), ou um **while** (mas para validar se **a = 0 e vet[0] = 0** eu precisaria lê-los antes de entrar no ciclo, o que poderia ser feito com um **if** bem colocado – inclusive fiz isso na questão Saltos Ornamentais – ); no final optei por um **while** com condição interna.

**scanf ("%d %s", &a, vet);**

**if (a == 0 && vet[0] == '0') break;**

Aqui eu leio os valores inserido pela entrada padrão (teclado) através da função **scanf,** e coloco-os nas variáveis **a** (que recebe o valor inteiro ao início) e em **vet** (que recebe o valor do contrato.

**for (i = 0, cont = 0; vet[i] != '\0'; i++){**

**aux2 = vet[i];**

**aux2 -= 48;**

**if (aux2 != a){**

**aux[cont] = vet[i];**

**cont++;**

**}**

**}**

Essa é a parte onde a primeira e segunda parte do algoritmo descrito no tópico 2 são realizadas. O ciclo **for** funciona a base do contador **i**, e através dele eu acessarei os diferentes elementos de **vet.** A cada ciclo eu atribuo à **aux2** o valor em **vet[i]**, **no entanto observe que vet[i] é um caractere e aux2, inteiro;** assim, apesar de vet[i] ser um dígito, aux2 não receberá o valor desse dígito, **mas sim sua posição na tabela ASCII**.

Assim o dígito 5, por exemplo, será recebido como 53. Verificando a tabela ASCII pode-se verificar que todos os dígitos estão em sequência da posição 48 até a 57, sendo assim para obter o valor matemático deles basta subtrair 48 de sua posição na tabela ASCII (de fato 53 – 48 = 5, por exemplo). Por isso, na linha subjacente, **aux2 sofre um decréscimo de 48**, para que se torne um número de fato.

Agora que aux2 é um número posso compará-lo com **a** (que é o dígito defeituoso e que foi lido como inteiro). Se **aux2 = a**, nada ocorre, mas se **aux2 for diferente de a** então **aux[cont]** recebe o valor de aux2. Observe duas coisas:

**1)** **aux será uma string formada somente por dígitos com valor diferente de a**

**2) aux é acesado por um contador próprio, o *cont*.**

O segundo ponto é importante, pois se eu usasse o mesmo contador **i** para aux então algumas posições de aux não teriam nada (ou melhor, teriam lixo). Por exemplo, se **i = 2 e o valor de vet[i] = a, aux[2] não receberia nada,** mas o contador passaria a valer 3 no próximo ciclo, e aux[2] de fato não seria modificado.

Esclarecido isso, observe também que toda vez que **aux[cont]** receber um conteúdo é fundamental que o contador **cont** passe para o próximo valor (por isso **cont++** na linha seguinte).

Isso poderia ter sido feito através de um operador pós-fixado também, como em **aux[cont++] = vet[i].**

Como eu disse antes, ao final **aux** será uma string formada somente pelos valores de **vet** diferentes de a. Mas para aux ser uma string de fato é preciso “tampar” ela, colocando um ‘\0’ no seu final. A posição final de aux será **cont,** ao fim do ciclo, pois ele sempre é acrescido de uma unidade quando **aux[cont]** é acessado; então quando o ciclo chegar ao final **cont** terá sido acrescido mas aux nessa posição não terá recebido nada (pois para ele ter recebido, **cont** teria que ser acrescido de novo).

Assim **aux[cont] = '\0'** “tampa” a string aux.

Resolvida a segunda parte do algoritmo, basta eliminar os zeros antes do primeiro dígito não nulo, conforme descrito nos tópicos 2 e 3.

**if (cont == 0) printf ("0\n");**

**else{**

**for (i = 0; aux[i] == '0'; i++);**

**if (i < cont) printf ("%s\n", aux + i);**

**else printf ("0\n");**

Aqui eu começo analisando se **cont** vale 0. Como ele não foi modificado após a execução do último bloco, **cont** contém o tamanho de aux. Se ele vale zero, então nunca ocorreu do valor de vet[i] ser diferente de “a”, logo **vet é formado somente por valores iguais a “a”**, o que faz com que o valor após a remoção seja imediatamente zero.

Simplificadamente, se cont = 0 então o valor final no contrato será zero, logo basta exibir 0 na tela.

Agora se **cont** é não nulo, então em algum momento vet[i] é diferente de a e portanto aux tem alguma coisa. Nesse caso basta eliminar os zeros do início e exibir o resultado; e é precisamente isso que eu faço através do ciclo **for** e no estilo comentado no tópico 3.

A linha de código **for (i = 0; aux[i] == '0'; i++)** vai aumentar o valor de **i** enquanto aux valer 0. Ao final, **i** será a posição de aux onde ocorre o primeiro dígito não nulo. Isso já é suficiente, pois eu percebo que **(aux + i)** é um ponteiro para o início de uma string **contida no aux original**, sendo que essa não possui zeros à esquerda. Logo, para exibir o resultado na tela basta usar o **printf ("%s\n", aux + i).**

Na verdade quase, existe ainda a possibilidade de aux ser formada somente por zeros, e nesse caso o método descrito não funciona. Mas se aux for formado somente por zeros, **i** ao final do for será exatamente igual a **cont (**tamanho de aux, e que até agora não foi modificado novamente). Assim, se **i < cont**, então aux não é formado somente por zeros e basta exibir **(vet + i)**. Já se essa condição não for satisfeita, então **i = cont** e o resultado deve exibido deve ser 0 (pois aux é formada somente por zeros).

**6) Observações finais:**

Nenhuma.

**7) Tópicos abordados na questão:**

**1)Fundamentos de C:** Estrutura geral, Diretivas de Pré-processamento e Função main.

**2)Fundamentos de C:** Dados primitivos, operadores matemáticos e relacionais, funções de entrada e saída de dados (printf e scanf)

**4)Estruturas de Controle de Fluxo:** Desvios (simples, composto, encadeado)

**5)Estruturas de Controle de Fluxo:** Laços (for, while)

**6)Estruturas de Controle de Fluxo:** Comandos Break;

**8)Strings:** Declaração, Inicialização e Manipulação;

**11)Ponteiros:** Definição, Vetores como ponteiros.

2552

PãodeQueijoSweeper

**1) Descrição do Problema:**

(Presente em <https://www.urionlinejudge.com.br/judge/pt/problems/view/2552> )

Está chegando a grande final do Campeonato Nlogonense de Surf Aquático, que este ano ocorrerá na cidade de Bonita Horeleninha (BH)! Nesta cidade, o jogo PãodeQueijoSweeper é bastante popular!

O tabuleiro do jogo consiste em uma matriz de **N** linhas e **M** colunas. Cada célula da matriz contém um pão de queijo ou o número de pães de queijo que existem nas células adjacentes a ela. Uma célula é adjacente a outra se estiver imediatamente à esquerda, à direita, acima ou abaixo da célula. Note que, se não contiver um pão de queijo, uma célula deve obrigatoriamente conter um número entre 0 e 4, inclusive.

Dadas as posições dos pães de queijo, determine o tabuleiro do jogo!

## **Entrada**

A entrada contém vários casos de teste. A primeira linha de cada caso contém os inteiros **N** e **M** (1 ≤ **N, M** ≤ 100). As próximas **N** linhas contém **M** inteiros cada, separados por espaços, descrevendo os pães de queijo no tabuleiro. O **j**-ésimo inteiro da **i**-ésima linha é 1 se existe um pão de queijo na linha **i** e coluna **j** do tabuleiro, ou 0 caso contrário.

A entrada termina com fim-de-arquivo (EOF).

**Saída**

Para cada caso de teste, imprima **N** linhas com **M** inteiros cada, não separados por espaços, descrevendo a configuração do tabuleiro. Se uma posição contém um pão de queijo, imprima 9 para ela; caso contrário, imprima o número cuja posição deve conter.

| Exemplo de Entrada | Exemplo de Saída |
| --- | --- |
| 4 4 0 0 1 1 0 1 0 1 0 0 1 0 1 1 0 1 1 2 0 1 | 0299 1949 1393 9939 19 |

Programação Competitiva, UFPR

**2)Algoritmo de Resolução:**

Funcionando como um campo minado “ao contrário”, aqui eu tenho uma matriz com zeros e uns, onde 1 significa que existe um pão de queijo naquele espaço e 0 que não existe.

Uma vez identificada a matriz, basta procurar os espaços onde tem 0 e colocar neles um número de 0 a 4, referente a quantidade de pães de queijo ao seu redor.

Existe apenas um detalhe, é preciso também colocar em cada posição onde existe um pão de queijo o número 9, isto é, trocar 1 e 9. É especialmente importante realizar essa substituição de maneira prévia e explicarei o porquê durante a explicação do código.

**3) Pensando no Algoritmo em Termos Computacionais:**

Enquanto o programa lê a matriz, se o elemento inserido for um ‘1’, ele automaticamente já o trocará por um ‘9’.

Após lida, basta varrer a matriz usando dois ciclos **for** (um move um contador através das linhas, e para cada linha um outro move um contador através das colunas). Para toda posição em que eu estiver, devo analisar os espaços ao redor e contar a quantidade de 9’s. Isso pode ser feito com um outro **contador**, que sofre um acréscimo de uma unidade se for identificado um pão de queijo ou permanece inalterado se não.

Em seguida o conteúdo dessa posição recebe o valor do contador, e o contador é anulado para ser usado na próxima posição novamente.

É fundamentalmente importante atentar para se o espaço analisado existe na matriz, isso é, se o programa não está tentando acessar uma posição que não pertence a matriz. Uma visão mais detalhada nisso será mostrada na explanação do código.

**4) Código-Fonte:**

**#include <stdio.h>  
  
main ()  
{  
 int N, M, i, j, p = 0;  
 while (scanf ("%d %d", &N, &M) != EOF){  
 int vet[N][M];  
  
 for (i = 0; i < N; i++)  
 for (j = 0; j < M; j++){  
 scanf ("%d", &vet[i][j]);  
 if (vet[i][j] == 1) vet[i][j] = 9;   
 }  
  
 for (i = 0; i < N; i++)  
 for (j = 0; j < M; j++)  
 if (vet[i][j] != 9) {  
 if (i > 0) if (vet[i-1][j] == 9) p++;  
 if (i < N - 1) if (vet[i+1][j] == 9) p++;  
 if (j > 0) if (vet[i][j-1] == 9) p++;   
 if (j < M - 1) if (vet[i][j+1] == 9) p++;   
 vet[i][j] = p;  
 p = 0;  
 }   
   
 for (i = 0; i < N; i++, printf ("\n"))  
 for (j = 0; j < M; j++)  
 printf ("%d", vet[i][j]);  
 }  
}**

**5) Explicação do Código-Fonte:**

O código começa com a estrutura geral de C: através da diretiva de pré-processamento #include eu adiciono ao programa o arquivo header stdio.h. Após isso eu construo a função main, fundamental em todo código C.

Em seguida ocorre a declaração da variáveis, através da linha de código **int N, M, i, j, p = 0,** aqui N e M são as dimensões da matriz, **i e J** são os contadores que usarei nos ciclos **for** e **p** é o contador de pães de queijo que mencionei no tópico 3.

**#include <stdio.h>**

**main ()**

**{**

**int N, M, i, j, p = 0;**

**while (scanf ("%d %d", &N, &M) != EOF){**

**(…)**

**}**

}

O ciclo while aqui declarado tem o objetivo de manter o código repetindo enquanto a função **scanf** conseguir ler informações. A questão diz que o fim do processamento é dado por EOF (End Of File), isto é, fim do arquivo. No site URI muitas vezes um arquivo com várias entradas é colocado como entrada para o código, nesse caso o programa deve parar de funcionar somente quando a função scanf não conseguir mais ler entrada alguma. Quando isso ocorre, a função retorna -1 e essa informação poderia ser usada para dar fim ao ciclo, no entanto a notação usada, **scanf ("%d %d", &N, &M) != EOF,** também é válida e funciona de maneira similar (enquanto a leitura do scanf for diferente de -1 – que é o fim de arquivo, ou EOF – o ciclo continua).

Repare que o scanf lê os valores referentes as dimensões da matriz. Usando esses valores (agora já lidos) eu construo a matriz de N por M, através da linha de código **int vet[N][M].**

**for (i = 0; i < N; i++)  
 for (j = 0; j < M; j++){  
 scanf ("%d", &vet[i][j]);  
 if (vet[i][j] == 1) vet[i][j] = 9;   
 }**

Nessa parte do código eu leio os elementos da matriz através de dois ciclos for, um responsável por mover a coluna e outro a linha, como mencionei no tópico 3. O comando scanf é quem lê a atribui à vet[ i ][ j ] (elemento aij da matriz) o valor lido. Imediatamente eu verifico se o valor lido é igual a 1 através da linha de código **if (vet[i][j] == 1) vet[i][j] = 9**, e como pode-se observar troco-o por 9 se for (usando a substituição prévia que mencionei no tópico 2).

**for (i = 0; i < N; i++)  
 for (j = 0; j < M; j++)  
 if (vet[i][j] != 9) {  
 if (i > 0) if (vet[i-1][j] == 9) p++;  
 if (i < N - 1) if (vet[i+1][j] == 9) p++;  
 if (j > 0) if (vet[i][j-1] == 9) p++;   
 if (j < M - 1) if (vet[i][j+1] == 9) p++;   
 vet[i][j] = p;  
 p = 0;  
 }**

Nessa parte do código eu efetuo a parte mais importante: colocar a quantidade de pães de queijo adjacentes nas casas que antem tinham 0. Para fazer isso eu varro o vetor novamente através de dois ciclos for, e analiso cada elemento.

Primeiramente verifico se ele não é um pão de queijo, isto é, se **vet[i][j] != 9.** Se não for, passo a contar os pães ao redor. Para isso tenho que acessar as posições **vet[i][j+1], vet[i][j-1], vet[i+1][j] e vet[i-1][j],** verificando se nelas existe o pãozinho.

Mas antes de fazer isso, como antes dito, é fundamental que eu verifique se essas posições existem (isto é, fazem parte da matriz). Ora, se 0 =< i < N e 0 =< j < M basta verificar se i > 0 para garantir que i – 1 existe e portanto **vet[i-1][j]** também. Já para **vet[i+1][j]** eu devo verificar se i + 1 < N (pois 0 < i < N), o que equivale a analisar se i < N – 1. De maneira semelhante, j > 0 para **vet[i][j-1]** existir e j < M – 1 para **vet[i][j+1]** existir.

Essa verificações são feitas através dos **if’s**. Uma vez garantida a existência da posição, verifico se nela existe pãozinho (isto é, seu valor é igual a 9), se existir eu adiciono uma unidade ao contador **p**. Esse processo continua até que eu termine de analisar todas as casas adjacentes, e ao fim eu atribuo ao valor na posição o valor do contador (que possui a quantidade de pãeszinhos adjacentes).

Observe que, como existem no máximo 4 pãezinhos adjacentes, o valor 9 nunca vai se referir a uma quantidade de pães (daí a grande importância de trocar 1 por 9 antes do início desse código, pois senão o número 1, que de início se refere a presença do pão de queijo, seria interpretado também como uma quantidade de pães adjacentes: uma confusão que faria tudo dar errado).

Após realizar a atribuição eu zero o contador de pães e passo para o próximo, repetindo o processo até construir a matriz exatamente a maneira pretendida.

**for (i = 0; i < N; i++, printf ("\n"))  
 for (j = 0; j < M; j++)  
 printf ("%d", vet[i][j]);**

Essa parte final simplesmente ultiliza a função printf para exibir todos os elementos da matriz esperada. Para obter a formatação correta observe que toda vez que ocorre uma passagem de linha (isto é, o contador i referente a linha é acrecido de uma unidade) o programa executa o comando **printf ("\n"),** que pula uma linha.

**6) Observações finais:**

Nenhuma.

**7) Tópicos abordados na questão:**

**1)Fundamentos de C:** Estrutura geral, Diretivas de Pré-processamento e Função main.

**2)Fundamentos de C:** Dados primitivos, operadores matemáticos e relacionais, funções de entrada e saída de dados (printf e scanf)

**4)Estruturas de Controle de Fluxo:** Desvios (simples, aninhado)

**5)Estruturas de Controle de Fluxo:** Laços (for, while)

**9)Matrizes:** Declaração, Inicialização e Manipulação.

2311

Saltos Ornamentais

**1) Descrição do Problema:**

(Presente em <https://www.urionlinejudge.com.br/judge/pt/problems/view/2311> )

Em uma determinada competição de saltos ornamentais, cada salto recebe um grau de dificuldade e é avaliado por sete juízes. Após cada salto, os juízes, que não se comunicam uns com os outros, mostram suas notas. Um salto é cotado entre zero e dez pontos. Depois de apresentadas as notas, a mais alta e a mais baixa são descartadas. O restante é somado e multiplicado pelo grau de dificuldade do salto, que gira entre 1,2 e 3,8, definido sempre antes do início da apresentação do atleta. O julgamento então é feito da seguinte forma: supondo que um saltador tenha sua nota de partida (seu grau de dificuldade de movimento) avaliada em 2,0 e tire notas 6,0, 5,0, 5,0, 5,0, 5,0, 5,0, 4,0 em sua execução. Disso, retira-se a nota mais baixa e a mais alta, o que gera um resultado parcial de 25,0. Então, pega-se a nota de execução e multiplica-a pela nota de partida para se chegar ao resultado final, que neste exemplo é de 50,0. Seu programa deve apresentar o resultado de uma competição de acordo com estas regras.

**Entrada**

A primeira linha de entrada contém o número de competidores **N** (0 ≤ **N** ≤ 100). A seguir são mostrados os nomes de cada um dos competidores seguidos pelo grau de dificuldade dos seus saltos **GD** (1.2 ≤ **GD** ≤ 3.8) e, a seguir, na linha seguinte, as 7 notas recebidas **N1 a N7** (0 ≤ **N1** a **N7** ≤ 10).

## **Saída**

A saída deve apresentar o resultado da competição, com o nome dos competidores seguido de seu resultado, na ordem em que os dados foram lidos.

| Exemplo de Entrada | Exemplo de Saída |
| --- | --- |
| 3 Gabriela 2.0 6.0 5.0 5.0 5.0 5.0 5.0 4.0 Marina 1.5 8.5 7.0 8.0 8.0 8.4 7.5 7.7 Mafalda 3.0 6.0 7.0 6.5 6.8 7.9 6.2 6.6 | Gabriela 50.00 Marina 59.40 Mafalda 99.30 |

**2)Algoritmo de Resolução:**

Vamos começar com uma fácil. O algoritmo para resolver esse problema que usarei é aquele descrito na própria questão:

-Pegar todas as notas

-Somar todas exceto a maior e a menor

-Exibir o nome da competidora com o produto da soma pelo grau de dificuldade

**3) Pensando no Algoritmo em Termos Computacionais:**

Já que essa era uma questão simples a minha ideia era utilizar diferentes conhecimentos obtidos ao longo da disciplina na resolução. Primeiro eu notei que não foi informado o tamanho máximo do nome entrado; assim eu poderia alocar uma quantidade grande de memória, como vet[99], só por segurança, ou poderia usar a função realloc para selecionar exatamente o tamanho correto, o que é bem mais interessante.

Agora para fazer o computador executar a sequência de passos mencionada eu pretendo ler as notas como um vetor, ordená-las através de um Bubble Sort e então somar os valores da posição 1 (segunda posição) até a posição 6 (penúltima posição), colocando tal soma numa variável Soma.

(Observe que, como o vetor foi ordenado, a primeira posição tem o menor valor e a última o maior, então eu vou ignorá-las assim como diz o algoritmo)

Dai basta exibir o nome da moça acompanhado pelo produto do grau de dificuldade com a Soma, simples assim.

Mas antes de seguir em frente, por que um Bubble Sort? Por que eu li sobre ele recentemente e queria fazer um, e também porque pensei em fazê-lo com uma função recursiva, o que agrega conteúdo ao trabalho.

(inclusive descobri esses algoritmos de ordenação enquanto fazia esse trabalho, e comentarei mais sobre essa descoberta ao longo das próximas questões).

Imagino que o professor tenha plena noção do que é um bubble sort, mas vou explicá-lo brevemente pois é parte da lógica da questão: o Bubble sort é um algoritmo de ordenação onde eu comparo um elemento com aquele após ele até chegar ao fim do vetor, e se meu objetivo é organizar em ordem crescente então devo garantir que aquele antes é menor que o posterior; se esse não for o caso, troco-os de posição.

Devo repetir esse algoritmo até que não ocorra nenhuma troca no vetor inteiro, pois isso ocorrerá quando ele tiver sido ordenado.

**4) Código-Fonte:**

**#include <stdio.h>**

**#include <stdlib.h>**

**void BubbleSort (float \*vet, int Tamanho)**

**{**

**int i, flag;**

**float aux;**

**for (i = 0, flag = 0; i < Tamanho - 1; i++)**

**if (vet[i] > vet[i + 1]){**

**aux = vet[i];**

**vet[i] = vet[i + 1];**

**vet[i + 1] = aux;**

**flag = 1;**

**}**

**if (flag == 1) BubbleSort (vet, 7);**

**else return;**

**}**

**main ()**

**{**

**int i, cont, T, N;**

**double Soma;**

**float dificuldade, vet[7];**

**char \*nome, aux, lixo;**

**scanf ("%d", &N);**

**for (cont = 0; cont < N; cont++){**

**/\* Usando Realloc para ler construir um vetor com precisamente o tamanho do nome \*/**

**scanf ("%c", &lixo);**

**i = 0;**

**T = 1;**

**nome = (char \*) malloc(sizeof(char));**

**do{**

**scanf ("%c", &aux);**

**if (aux != '\n') nome[i++] = aux;**

**if (i >= T)**

**nome = (char \*) realloc(nome, sizeof(char)\*++T);**

**} while (aux != '\n');**

**nome[i] = '\0';**

**/\* Lendo a dificuldade da prova e um vetor com as notas \*/**

**scanf ("%f", &dificuldade);**

**for (i = 0; i < 7; i++) scanf ("%f", &vet[i]);**

**/\*Organizo o vetor e ignoro a primeira e última posição \*/**

**BubbleSort(vet, 7);**

**for (i = 1, Soma = 0; i < 6; i++) Soma += vet[i];**

**printf ("%s %.2lf\n", nome, Soma\*dificuldade);**

**free(nome);**

**}**

**}**

**5) Explicação do Código-Fonte:**

Inicialmente faço o de sempre: incluo as bibliotecas stdio.h e stdlib.h através da diretiva de pré-processamento #include e construo a função main (na verdade eu faço uma outra função antes disso, a BubbleSort, que é responsável por executar o algoritmo de mesmo nome)

De início declaro algumas variáveis:

**int i, cont, T, N;**

**double Soma;**

**float dificuldade, vet[7];**

**char \*nome, aux, lixo;**

i e cont serão contadores. T será usado para alocar o tamanho certo para o nome entrado. N é o número de casos de teste. Soma é, bom, a soma antes mencionada. Dificuldade é o grau de dificuldade da prova e vet é um vetor que recerá as notas obtidas pela senhorita. Nome é um ponteiro de caractere que usarei para colocar o nome da moça, aux é um auxiliar que será importante na hora do alocamento de memória e lixo será usado como para limpar o “\n” inserido após o nome da garota.

Certo, eu poderia usar um limpador de buffer para isso, e de fato foi o que eu fiz primeiro. Contudo, todas as vezes que eu submetia o código ele dava 20% de erro, e depois de bater muita cabeça um amigo me aconselhou a remover o setbuf(stdin, NULL), e dai coloquei um scanf que lia o “\n” no lugar, e funcionou. Não sei exatamente por que isso ocorreu, mas de fato a única coisa que mudei foi o setbuf.

No entanto, vamos voltar a questão:

**scanf ("%d", &N);**

**for (cont = 0; cont < N; cont++){**

**/\* Usando Realloc para ler construir um vetor com precisamente o tamanho do nome \*/**

**scanf ("%c", &lixo);**

**(...)**

**}**

Em seguida eu leio a quantidade N de casos de teste. No entanto, o “\n” inserido após esse número é incomodo, então eu utilizo outro scanf para lê-lo e colocá-lo numa variavel lixo, que servirá somente para isso mesmo.

O ciclo **for** é responsável por executarf o código para cada caso de entrada, assim ele move o contador cont até que ocorram N ciclos (o que acontecerá quando cont = N – 1).

Em seguida eu ultilizo uma alocação dinâmica de memória para construir o vetor nome com o tamanho exato da string inserida, veja:

**i = 0;**

**T = 1;**

**nome = (char \*) malloc(sizeof(char));**

**do {**

**scanf ("%c", &aux);**

**if (aux != '\n') nome[i++] = aux;**

**if (i >= T)**

**nome = (char \*) realloc(nome, sizeof(char)\*++T);**

**} while (aux != '\n');**

**nome[i] = '\0';**

Aqui eu aloco, de início, somente uma posição de memória para nome. Em seguida meu código lerá o primeiro caractere da string inserida pelo usuário, e em seguida eu verifico se esse é um “\n”, pois eu não gostaria que o “\n” inserido após a string fosse lido como parte dela. Se não for, então atribuo à posição de memória nome + i o conteúdo desse aux (observe que \*(nome + i) = nome[i]) e em seguida aumento i em uma unidade.

Terminado esse if, verifico se o valor de i é maior ou igual a T, que é o tamanho da string. Observe que inicialmente T vale 1, que é o tamanho mínimo que essa string deve ter. Se i se tornou igual a T, então na próxima leitura o vetor nome não terá tamanho para receber esse dado.

A solução é realocar o espaço destinado a nome, e para isso eu aumento o valor de T (através do operador “++” pré-fixado) e uso o comando realloc, atribuindo a posição de memória resultante para o ponteiro nome novamente.

Desse modo eu sempre irei ler um caractere, e se ele não for o “\n” (que marcará o fim da string” eu irei coloca-lo na string nome, que terá livre somente um espaço; por isso em seguida eu aloco mais um espaço livre imaginando que pode haver outro caractere a ser lido.

Note que por essa lógica nome sempre terá o espaço de índice igual a i vazio. Assim depois que eu termino de ler a string (um “\n” é inserido) coloco um “\0” na posição num + i (a qual está vazia) e num passa a ser uma string perfeita.

Seguindo para a próxima parte do código, eu leio a dificuldade (um float) e coloco seu valor na variavel de mesmo nome. O ciclo **for** é usado para preencher o vetor de floats vet com as notas da competidora.

**/\* Lendo a dificuldade da prova e um vetor com as notas \*/**

**scanf ("%f", &dificuldade);**

**for (i = 0; i < 7; i++) scanf ("%f", &vet[i]);**

**/\*Organizo o vetor e ignoro a primeira e última posição \*/**

**BubbleSort(vet, 7);**

Na linha seguinte eu desejo ordenar o vetor vet, e para isso chamo a função que construi, a Bubble Sort (observação: eu construi essa função sozinho, baseado no que entendi do Bubble Sort, e não simplesmente copiei ela da internet).

A função recebe dois argumentos: o ponteiro para o início do vetor que será ordenado (no caso, vet) e o tamanho do vetor (que é 7, como informa o enunciado do problema). Vamos ver agora como eu fiz o Bubble Sort:

**void BubbleSort (float \*vet, int Tamanho)**

**{**

**int i, flag;**

**float aux;**

**for (i = 0, flag = 0; i < Tamanho - 1; i++)**

**if (vet[i] > vet[i + 1]){**

**aux = vet[i];**

**vet[i] = vet[i + 1];**

**vet[i + 1] = aux;**

**flag = 1;**

**}**

**if (flag == 1) BubbleSort (vet, 7);**

**else return;**

**}**

Uma vez recebidos os paramêtros formais da função, passo a declarar i, que será um contador, e flag, um gatilho o qual indicará se houve ou não alguma troca ao longo do ciclo antes executado.

Para permitir a busca repetitiva descrita no algoritmo do bubble sort eu utilizo uma função recursiva, e para fazer as comparações entre elementos sucessivos eu ultilizo o ciclo **for** dentro dela. A cada ciclo for eu comparo vet[i] e o elemento posterior, vet[i + 1], trocando-os de posição se o primeiro for maior que o segundo. Esse for deve perdurar a medida que existir um elemento posterior, o que ocorrerá enquanto i < Tamanho – 1 (pois assim o maior valor que i terá é de Tamanho – 2, e seu sucessor será Tamanho – 1, que é a última posição válida de vet).

Todas as vezes que ocorre alguma troca a flag recebe o valor 1, que informa esse fato para o if após o ciclo for e assim permite que a própria função seja chamada outra vez. Quando não houver troca alguma, o que significa que o vetor já está ordenado, a flag não é modificada e o caso else leva a um return, finalizando a execução da função. (observe que no início do for a flag recbe o valor 0)

Agora passamos para as linhas finais do código:

**for (i = 1, Soma = 0; i < 6; i++) Soma += vet[i];**

**printf ("%s %.2lf\n", nome, Soma\*dificuldade);**

**free(nome);**

Através do ciclo **for** eu acresço o valor da variável soma (observe que ela recebe valor 0 no início do for, evitando que um valor anterior seja re-utilizado). No entanto, adiciono a ela somente os elementos de posição 1a 6 no vetor, para excluir o maior e menor.

Uma vez que soma tenha sido guardada, exibo na tela o nome da garota (guardada em nome – e que pode ser exibida como uma string, pois coloquei um “\0” no final dela – ) e o produto de soma pela dificuldade, com somente 2 casas decimais (como especificado pela questão).

Agora libero o espaço de memória previamente alocado, evitando acumular lixo no heap (já que o código vai ser executado N vezes).

**6) Observações finais:**

Nenhuma.

**7) Tópicos abordados na questão:**.

1403

Meu Avô é Famoso

**1) Descrição do Problema:**

(Presente em: <https://www.urionlinejudge.com.br/judge/pt/problems/view/1403> )

A família toda ficou excitada pela novidade. Todos sabiam que o meu avô tinha sido um excelente jogador de bridge por décadas, mas quando foi anunciado que ele estaria no Guinness Book, o livro dos recordes, como o jogador de bridge de maior sucesso de todos os tempos, wow, aquilo foi surpreendente.

A Associação Internacional de Bridge (AIB) tem mantido, por diversos anos, um ranking semanal dos melhores jogadores do mundo. Considerando que cada aparição em um ranking semanal constitui um ponto para o jogador, meu avô foi nominado o melhor jogador de todos os tempos porque ele conseguiu o maior número de pontos.

Tendo muitos amigos que também estavam competindo com ele, meu avô está extremamente curioso para saber que jogador(es) ficou(aram) com o segundo lugar. Ele precisa de um programa, o qual, dada uma lista com os ranking semanais, descubra que jogador(es) ficou(aram) com o segundo lugar, de acordo com o número de pontos.

**Entrada**

A entrada contém diversos casos de teste. Jogadores são identificados por inteiros de 1 a 10000. A primeira linha de um caso de teste contém dois inteiros **N e M**, indicando, respectivamente, o número de rankings disponíveis (2 ≤ **N** ≤ 500) e o número de jogadores em cada ranking (2 ≤ **M** ≤ 500). Cada uma das próximas **N** linhas contém a descrição de um ranking semanal. Cada descrição é composta por uma sequência de **M** inteiros, separados por um espaço em branco, identificando os jogadores que apareceram naquele ranking semanal. Você pode assumir que:

* em cada caso de teste há exatamente um melhor jogador e ao menos um segundo melhor jogador,
* cada ranking semanal consiste de **M** jogadores distintos.

O final da entrada é indicado por **N = M = 0**.

**Saída**

Para cada caso de teste da entrada seu programa deve produzir uma linha de saída, contendo o identificador do jogador que é o segundo melhor, em número de aparições nos rankings. Se há um empate para segundo lugar, imprima os identificadores de todos os segundo colocados, em ordem crescente. Cada identificador produzido deve ser seguido por um espaço em branco.

| Exemplo de Entrada | Exemplo de Saída |
| --- | --- |
| 4 5 20 33 25 32 99 32 86 99 25 10 20 99 10 33 86 19 33 74 99 32 3 6 2 34 67 36 79 93 100 38 21 76 91 85 32 23 85 31 88 1 0 0 | 32 33 1 2 21 23 31 32 34 36 38 67 76 79 88 91 93 100 |

ACM/ICPC South America Contest 2004.

**2)Algoritmo de Resolução:**

O algoritmo para resolver esse problema informalmente é bem simples. Basta contar, para cada número diferente, a quantidade de vezes em que ele aparece. Uma vez terminada essa etapa, devo ignorar aquele que aparece mais vezes e exibir os que apareceram uma quantidade de vezes igual a segundo maior quantidade de vezes na qual qualquer um apareceu.

No entanto, como fazer isso em código?

**3) Pensando no Algoritmo em Termos Computacionais:**

Esse é um dos interessantes casos onde é fácil encontrar, como humano, a resposta. Mas coordenar o computador para fazer isso já é outra história.

É claro que eu devo analisar quantas vezes cada número aparece, mas uma busca sequencial com base no que me é dado na entrada seria algo muito complicado, afinal, se a entrada tivesse M elementos, eu teria gastar M\*M ciclos para contar quantas vezes cada valor aparece.

Minha ideia então é organizar de modo crescente a entrada, pensando nela como um vetor. Uma vez organizada, eu pego cada valor e comparo com os próximos “n” até que encontre um diferente dele; observe que como eu ordenei o vetor, todos os elementos iguais ao que eu peguei estarão após ele (se eu começar da primeira posição).

Eu então guardo a quantidade “n” de valores iguais ao valor analisado e passo o contador para a posição “n + 1”, onde existe o primeiro diferente daquele, e repito o procedimento, comparando-o com os próximos “n”, guardando o número de igualdades e passando para o próximo.

No entanto, como associar o valor à quantidade de vezes que ele aparece?

Bom, uma solução criativa que me veio a mente foi criar uma Struct com dois campos, um para o número do jogador e outro para a quantidade de vezes em que ele aparece. E isso funcionou extremamente bem.

Se eu criar um vetor de “c” elementos, onde “c” é a quantidade de conjuntos com números de jogadores disjuntos, posso encontrar a maior e a segunda maior quantidade na qual um jogador aparece e então exibir somente o número dos jogadores cuja quantidade de vezes em que apareceram é igual à segunda maior quantidade mencionada. Vamos para o código!

**4) Código-Fonte:**

**#include <stdio.h>**

**#include <stdlib.h>**

**main ()**

**{**

**typedef struct{**

**int jogador;**

**int quant;**

**} Ficha;**

**Ficha \*num, aux2;**

**int N, M, i, j, \*vet, aux, flag, gap;**

**do{**

**scanf ("%d %d", &N, &M);**

**if (!N && !M) break;**

**vet = (int \*) malloc(sizeof(int)\*N\*M);**

**num = (Ficha \*) calloc(N\*M, sizeof(Ficha));**

**/\* Lendo os elementos da matriz \*/**

**for (i = 0; i < N\*M; i++)**

**scanf ("%d", &vet[i]);**

**/\* Ordenando o vetor usando um Comb Sort\*/**

**gap = N\*M;**

**do {**

**if (gap != 1) gap /= 1.3;**

**for (i = 0, j = gap, flag = 0; j < N\*M; i++, j++)**

**if (vet[i] > vet[j]){**

**aux = vet[i];**

**vet[i] = vet[j];**

**vet[j] = aux;**

**flag = 1;**

**}**

**} while (!(flag == 0 && gap == 1));**

**/\* Contando a quantidade de vezes que cada número aparece \*/**

**for (i = 0, j = 0; i < N\*M; i++){**

**if (!i) num[j].jogador = vet[i];**

**else{**

**if (vet[i] == num[j].jogador) num[j].quant++;**

**else num[++j].jogador = vet[i];**

**}**

**}**

**/\*Encontrando a maior e a segunda maior quantidade de vezes que um jogador aparece\*/**

**for (i = 0, M = -1, N = -1; i <= j; i++){**

**if (num[i].quant > M){**

**N = M;**

**M = num[i].quant;**

**}**

**else if (num[i].quant != M && num[i].quant > N) N = num[i].quant;**

**}**

**for (i = 0; i <= j; i++) if (num[i].quant == N) printf ("%d ", num[i].jogador);**

**printf ("\n");**

**}while (1);**

**}**

**5) Explicação do Código-Fonte:**

Começo o código ultilizando a diretiva de pré-processamento #include para adicionar ao meu código os arquivos headers stdio.h e stdlib.h. O primeiro é essencial para a entrada e saída de dados, enquanto o segundo é importante por me permitir usar as funções malloc e calloc.

Início a função main normalmente e declaro algumas variaveis, veja:

**typedef struct{  
 int jogador;  
 int quant;  
 } Ficha;  
   
 Ficha \*num, aux2;  
 int N, M, i, j,\*vet, aux, flag, gap;**

Através do comando **typedef** eu crio o tipo **Ficha**, que é a **struct** mencionada. Um vetor de fichas **\*num** será usado para armazenar os diferentes jogadores, mas como não sei seu tamanho vou criar apenas um ponteiro. O mesmo vale para o vetor onde ficarão os números dos jogadores, que é **vet**.

**N e M** são os números mencionados no enunciado, referentes ao número de rankings e de semanas. **I e j** são contadores. **Aux e aux2**, valores auxiliares que serão usados numa certa parte do código. **Gap e flag** serão usados no algoritmo de ordenação do vetor.

**do{   
 (…)**

**}while (1);**

A ideia é executar um ciclo, gerando resultado para cada caso de teste, até que N = M = 0. Assim eu crio um ciclo **do...while** “infinito”, que será finalizado por um **break** interno, que será ativado quando as entradas forem ambas nulas.

Eu inicio o código lendo ambas N e M. Imediatamente após essa leitura, faço uma validação de seus valores: se ambas forem nulas, então o ciclo é finalizado. Feita a verificação, devo alocar, para os ponteiros, o espaço de memória que eu preciso que eles tenham; para isso eu me valho das funções malloc e calloc:

**vet = (int \*) malloc(sizeof(int)\*N\*M);  
 num = (Ficha \*) calloc(N\*M, sizeof(Ficha));**

A razão pelo qual aloquei o espaço de **num** usando calloc ficará clara mais a frente. Após isso, leio os dados de entrada para o vetor **vet** , que são os rankings semanais. Para isso me valho de um ciclo **for,** movendo o contador **i** para que eu leia e coloque o conteúdo em cada elemento de **vet.**

**/\* Lendo os elementos da matriz \*/  
 for (i = 0; i < N\*M; i++)  
 scanf ("%d", &vet[i]);**

Agora chega uma parte interessante. Na primeira vez que eu submeti esse código recebi um Time Limit Exceeded, mas eu tinha uma noção do porquê isso tinha acontecido. O algoritmo de ordenação que eu estava usando era o mesmo da questão Elfo das Trevas, mas aqui é possível que N e M valham, cada, 500, o significaria 250000 espaços em **vet.**

Eu não entendo muito de algoritmos de complexidade, mas sei com aquele algoritmo uma ordenação dessas seria absurdamente longa (já que, por exemplo, o elemento 200001 seria comparado com todos os 200000 anteriores, e o 200002 seria comparado com todos os 200001 anteriores, o que me parece bem demorado – pelo pouco que sei, eu diria que esse é um algoritmo de complexidade O(n), já que a adição de um elemento a mais no vetor aumenta a quantidade de ciclos em “n”, onde “n” é a quantidade de elementos atual).

Felizmente, após fazer Elfo das Trevas, o nome Algoritmo de Ordenação (que eu montei de cabeça mesmo) me deixou curioso para saber se não existiriam outros desse tipo. Uma breve pesquisa me fez descobrir que existiam vários, dentre eles o Comb Sort, um dos mais ágeis. Eu o implementei no meu código com as linhas abaixo:

(apesar de tê-lo visto na internet, eu não copiei e colei nada de lá. Simplismente peguei a ideia e fiz do meu jeito)

**/\* Ordenando o vetor usando um Comb Sort\*/  
 gap = N\*M;  
 do {  
 if (gap != 1) gap /= 1.3;  
 for (i = 0, j = gap, flag = 0; j < N\*M; i++, j++)  
 if (vet[i] > vet[j]){  
 aux = vet[i];  
 vet[i] = vet[j];  
 vet[j] = aux;  
 flag = 1;  
 }  
 } while (!(flag == 0 && gap == 1));**

Eu imagino que o professor tenha pleno conhecimento desse algoritmo, mas vou comentá-lo brevemente: a ideia aqui é comparar dois elementos separados por uma distância igual a **gap** (**salto**, do inglês). Esse gap é calculado, no início, como o quociente da divisão do comprimento do vetor por 1.3. Nas próximas etapas, ele será igual ao quociente de seu valor anterior por 1.3 até que valha 1, nesse caso seu valor permanecerá imutável.

Ao comparar dois elementos separados por um **gap** de distância, devo organizá-los da maneira a qual pretendo que fiquem (isto é, como quero uma ordem crescente, aquele que se encontra antes deve ser menor do que se encontra depois; se esse não for o caso, troco-os de posição).

Esse procedimento deve prosseguir até que não ocorram mais trocas e gap seja igual a 1.

Inicialmente **gap** recebe o tamanho de **vet,** que é N\*M. Para construir o Comb Sort eu me utilizo de um ciclo que se repete enquanto não for verdade que **gap** é igual a 1 e o gatilho **flag** (responsável por me informar se houve alguma troca) informe 0 (houve troca).

Uma vez dentro do ciclo, verifico de gap é diferente de 1. Se sim, então reduzo seu valor ao dividi-lo por 1.3 (observe que, no primeiro ciclo, gap vai receber N\*M/1.3, como esperado). Após isso crio um ciclo **for** no qual eu movo duas váriaveis **i e j**, uma responsável pela primeira posição no vetor que se deseja analisar e a outra pela posição um “**gap”** a frente.

Então passo a comparar o conteúdo de **vet** nessas duas posições, e se o conteúdo que vem antes (vet[i]) for maior que aquele que vem depois (vet[j]), troco-os de posição (nessa troca eu uso o auxiliar **aux**)**.** Observe que se uma troca ocorrer, **flag** recebe o valor 1, o que significa que houve troca no último ciclo. (no entanto no início desse **for** ela recebe o valor 0, que permanecerá inalterado se não houve troca alguma – o que significa que ela de fato informa se houve ou não troca – ).

Dessa maneira eu ordeno completamente o vetor.

**/\* Contando a quantidade de vezes que cada número aparece \*/  
 for (i = 0, j = 0; i < N\*M; i++){  
 if (!i) num[j].jogador = vet[i];  
 else{  
 if (vet[i] == num[j].jogador) num[j].quant++;  
 else num[++j].jogador = vet[i];  
 }  
 }**

Hora de fazer o que prometi no tópico 3. A lógica aqui é simples: através do **for** eu movo o contador **i** e assim acesso os elementos de **vet** (agora ordenados), e acesso os elementos do vetor de fichas **num** com o contador **j.** Ora, se i = 0, então devo começar fazendo num[j].jogador receber o número do jogador (vet[i]) (que é o realizado pelo if de início), agora vamos ver por quanto tempo vet[i] continua tendo o mesmo valor. Para isso eu lido com o caso else, observe que quando ele ocorrer a variavel i já vei ter sido acrescida de uma unidade, passando a valer 1.

Agora devo verificar se esse novo elemento de vet é igual ao número do jogador que já guardei (num[j].jogador). Se for, então o a quantidade de vezes que esse número apareceu (num[j].quant) deve ser acrescida de uma unidade; se não, então devo passar para o próximo elemento de num, pois vet tem um jogador diferente.

Dessa maneira eu vou guardar a quantidade de vezes que cada jogador aparece, pois enquanto vet[i] continuar sendo igual ao jogador em num[j], a quantidade de vezes que o sujeito aparece vai aumentando.

Observe então a importância do calloc. Como ele já deixa cada posição de num com 0, eu posso simplesmente aumentar o valor na posição num[j].quant sem me preocupar em anular essa casa primeiro (pois ela já foi anulada). Atente também para o fato de que, na linha **num[++j].jogador = vet[i],** eu primeiro aumento j (operador pré-fixado) e em seguida coloco algum conteúdo em num[j].jogador, o que faz com que j sempre se refira a uma posição de num que já possui conteúdo. Essa informação será importante em seguida.

**/\*Encontrando a maior e a segunda maior quantidade de vezes que um jogador aparece\*/**

**for (i = 0, M = -1, N = -1; i <= j; i++){**

**if (num[i].quant > M){**

**N = M;**

**M = num[i].quant;**

**}**

**else if (num[i].quant != M && num[i].quant > N) N = num[i].quant;**

**}**

Agora basta encontrar a segunda maior quantidade de vezes que um jogador apareceu. Para isso vou reutilizar as variáveis N e M, já que não vou mais precisar delas; aqui N será o segundo maior valor de aparições e M o maior.

Para encontrar esses valores eu uso um ciclo para percorrer num, atentando somente aos elementos quant. De início coloco M = -1 e N = -1, o que garante que eles recebam algum conteúdo na primeira vez que forem comparados. (pois o número de aparições deve ser maior ou igual a 0).

Aqui eu usarei o fato de que j não foi modificado para colocá-lo como margem limite do meu contador i. Observe que existe conteúdo em num[j] (como expliquei há pouco), então devo levar em consideração também o caso onde i = j (daí i <=j como condição do **for).**

A cada ciclo eu encontrarei a posição de num[i].quant em relação a M e N. Existem somente 5 possibilidades:

**1)** **num[i].quant > M**

**2) num[i].quant = M**

**3) M > num[i].quant > N**

**4) num[i].quant = N**

**5) N > num[i].quant**

Caso a primeira ocorra, eu desejo que **M** receba o valor de **num[i].quant** e **N** (segundo maior) receba o valor que **M** tinha. Isso é garantido pelo primeiro if dentro do for.

Caso o segundo ocorra, eu gostaria que nada ocorresse, pois N = M é algo que eu jamais queira que ocorra (por isso, no if encadeado com o else, uma das condições é num[i].quant != M).

No terceiro caso eu gostaria que M permanece inalterado, mas N receberia num[i].quant. O caso if encadeado com o else é o responsável por gerar isso (pois o else foi acessado, então num[i].quant é menor ou igual a M. Se o if encadeado com o else for acessado, então num[i].quant não é igual de M e maior que N, e portanto só pode ser menor que M e maior que N)

Já os casos 4 e 5 me são completamente desinteressantes, fazendo com que eu queira que nada aconteça quando eles ocorrerem. E de fato eles não atendem condição nenhuma no dos if’s, acabam não fazendo nada mesmo.

Ao final desse processo eu terei o valor da segunda maior quantidade de vezes que um jogador aparece, que é N. Com isso, e lembrando que os jogadores estão em ordem crescente (pois quando eu os passei de vet para num eu fui passando um em um na ordem,o que manteve a organização antes feita).

Assim, para apresentar o resultado em ordem crescente basta vasculhar todos os num[i] e exibir somente o num[i].jogador daqueles cujo num[i].quant for igual a N. E é exatamente isso que faço com a linha de código abaixo:

**for (i = 0; i <= j; i++) if (num[i].quant == N) printf ("%d ", num[i].jogador);**

Com isso está resolvido o programa. Coloco um printf (“\n”) final somente para exibir as saídas com formatação adequada e o ciclo recomeça, podendo ser finalizado se N = M = 0. Observe que todas as variáveis são “limpadas” antes de serem usadas: i e j recebem 0 no início dos ciclos, M e N recebem -1 no início do ciclo de busca pelo maior e segundo maior quantidades e etc. Assim nada do ciclo anterior se mantém e o programa funciona corretamente.

**6) Observações finais:**

Nenhuma.

**7) Tópicos abordados na questão:**

1726

Amigos

**1) Descrição do Problema:**

(Presente em <https://www.urionlinejudge.com.br/judge/pt/problems/view/1726> )

Você quer planejar uma grande festa de aniversário com seus amigos. Durante o planejamento você percebeu que você deve fazer inúmeros operações com conjuntos de amigos. Existe um grupo que consiste do Arthur, Biene e Clemens. Existe outro grupo de amigos que você conhece do snowboarding que consiste do Daniel, Ernst, Frida e Gustav. Se você quer convidar ambos, o resultado do grupo da festa consiste de **g1 + g2** (o resultado é a união de ambos os grupos). Então você pode computar a intersecção dos dois grupos **g1 \* g2**, que consiste no conjunto vazio. Talvez você queira convidar o grupo g1, mas excluindo todos os membros do outro grupo g2, que pode ser escrito como **g1 – g2**. Intersecção (\*) precede sobre união (+) e diferença (-). Todas as operações são associadas a esquerda, o que significa que em **A op1 B op2 C** você primeiro deve avaliar **A op1 B** (desde que op1 e op2 possuam uma **precedência igual**).

## **Entrada**

A entrada consiste de uma ou mais linhas. Cada linha contém uma expressão que você deve avaliar. Expressões são sintaticamente corretas e somente consistem dos seguintes caracteres:

* '{' e '}'
* Os elementos 'A' à 'Z' significando amigos de Arthur até Zora.
* Operações '+', '-' e '\*'
* '(' e ')' para agrupar operações
* Caracter de nova linha '\n' marcando o fim de uma expressão.

Uma linha nunca é maior que 255 caracteres.

## **Saída**

Como saída, mostre o conjunto de resultados entre chaves ‘{’ e ‘}’, cada um em uma linha. Imprima os elementos de cada conjunto em ordem alfabética.

| Exemplo de Entrada | Exemplo de Saída |
| --- | --- |
| {ABC}  {ABC}+{DEFG}+{Z}+{}  {ABE}\*{ABCD}  {ABCD}-{CZ}  {ABC}+{CDE}\*{CEZ}  ({ABC}+{CDE})\*{CEZ} | {ABC}  {ABCDEFGZ}  {AB}  {ABD}  {ABCE}  {CE} |

University of Ulm local Contest 1999/2000

**2)Algoritmo de Resolução:**

Essa deu trabalho. Pensar em um algoritmo para resolver expressões desse tipo não é tão difícil, mas eu tive que segmentar minha visão sobre a resolução desse problema para chegar a algum resultado.

Comecei pensando na seguinte sequência de passos:

-Localizar o parêntese mais interno

-Procurar uma intersecção e resolvê-la (repetir isso até que não seja encontrada nenhuma intersecção)

-”Andando” da esquerda para a direita, resolver os operadores que encontrar (que só devem ser **“+”** ou **“-”** , pois todos **“\*”** já foram resolvidos na segunda etapa do algoritmo)

-Localizar o parêntese mais interno (...)

Ao pensar nisso eu notei que teria que criar um ciclo para encontrar parênteses, e então resolver o que tinha dentro, e dai procurar parêntese, e assim vai.

Passei logo a construir o código, mas acabei me perdendo em meio a tanta informação. Depois de apagar todo o código e ir dormir, acordei com uma visão mais razoável do problema.

Pensei que se eu conseguisse resolver uma linha sem parênteses, poderia posteriormente pensar em como resolvê-la com parêntese. Na realidade rapidamente cheguei a conclusão de que, se eu tivesse uma “função mágica” chamada ResolvedorDeLinha, que fosse capaz de resolver uma linha sem parênteses, eu poderia seguir a sequência de passos abaixo para resolver o problema:

-Localizar o parêntese mais interno

-Aplicar o ResolvedorDeLinha

(Repetir até não encontrar mais parêntese interno)

-Por fim, aplicar o Resolvedor de Linha uma última vez, chegando ao resultado.

Assim decidi criar um arquivo novo e me dedicar somente a construir o resolvedor de linha. (Talvez falando assim esse algoritmo pareça igual ao anterior, mas na hora de colocar na prática pensar em tantas manipulações sobre a entrada me deixou confuso)

Passemos então a entender como o Resolvedor de Linha deve funcionar:

-Localiza a operação \* e resolve (repetir até não encontrar \* novamente)

-Resolver todas as operações que encontrar seguindo da esquerda para a direita (que vão ser + ou -)

Okay, com isso em mente, vamos pensar em como colocar isso em código.

**3) Pensando no Algoritmo em Termos Computacionais:**

Para construir o ResolvedorDeLinha eu teria que varrer a entrada em busca de um **“ \* ”**, que pode ser executado com um simples **for** e um **if** pertinente (busca simples); é importante também criar uma *flag*, um indicador que me diga se no ciclo antes executado algum **“ \* ”** foi encontrado: se sim, há chance de ainda existirem outros e por isso devo executar a busca novamente; se não, então não há mais nenhuma intersecção para ser resolvida e poderei seguir para o próximo passo.

Agora eu devo procurar na string restante por um **+ ou -**. O que eu encontrar primeiro eu resolvo e repito até não haver mais nenhum dos dois. Ao final desse processo eu terei o resultado da linha.

Assim, unindo esse ResolvedorDeLinha com alguns detalhes, eu poderia resolver também uma expressão com parênteses. Contudo falta um detalhe sobre o ResolvedorDelinha, como ele resolverá as operações?

Vamos começar pensando na União, que é a mais simples. Além de varrer o *array* e encontrar um **“ + ”** , eu precisarei também localizar a posição do último **“ { “** e do próximo **“ { “.** Assim terei a posição inicial do conjunto antes do **+** (começa na posição após o primeiro **“ { “** e termina na posição antes do próximo **“ } ”** ) e daquele após ele (começa após o próximo **“ { “** e termina no primeiro **“}”** após ele). Posso então colocá-los em dois auxiliares **aux1 e aux2** para facilitar a manipulação**.**

Uma vez tendo **aux1 e aux2, a** união pode ser feita comparando aux2 a aux1; se o elemento analisado em aux2 não estiver em aux1 eu o adiciono **por cima da string original**, a partir do final de aux1, se estiver então nada faço (pois os conjuntos não devem ter letras repetidas – afinal, pra que colocar o nome de uma pessoa duas vezes num grupo?-)

Por exemplo:

{ABC}+{BDE}-{D}

(**aux1 = ABC** e **aux2 = BDE**)

{ABC**DE**{BDE}-{D}

(coloco os elementos de aux2 que não estão em aux1 por cima da string original, a partir do final de aux1 – posição onde estava o primeiro **“ } ”** – ).

{ABCDE**}**BDE}-{D}

(Agora eu coloco um **“ } ”** **para finalizar o conjunto união** e limpo tudo após esse **“ } ”** até que encontre outro **“ } ”**; sendo ele o último elemento que elimino– para evitar limpar outras operações ou conjuntos – ).

Após essa limpeza, resta **{ABCDE} -{D}**

(Com “limpar”, eu quero dizer simplesmente que troco o conteúdo daquela posição por um espaço vazio)

É importante mencionar que, após a adição de aux2, **haverá um e somente um “ } “ após aquele que finaliza união**. Pois a adição de dois conjuntos nunca vai ser menor do que o primeiro conjunto (o que é necessário para que pudesse haver um “**}”** antes do **“}”** que finaliza a união).

No caso da diferença eu farei algo parecido. Tendo as posições do último **“{“**  e do próximo **“{“** guardadas, eu coloco tais conjuntos em **aux1 e aux2** da mesma maneira que na União. Agora eu adiciono, por cima da string original e **começando pelo primeiro elemento de aux1** (e não pelo elemento final, como fiz na união) os elementos de aux1 que não estão em aux2. Dessa vez devo atentar para o fato de que podem existir **dois “{“** após o **“{“** que finaliza a diferença. Fato esse que só ocorrerá se, em algum momento, aux1 tiver um elemento que aux2 tem, o que leva a efetuação da diferença (e assim o conjunto diferença deve ser menor que **aux1**).

Assim eu posso criar *flag* simples que me diga se em algum momento essa diferença ocorreu: se sim, então existirão **dois “{“** após o **“{“** final e eu preciso limpar tudo até encontrar o segundo; se não, então existe somente um (pois o **“{“** que finaliza a diferença vai coincidir com o que finalizava aux1) e portanto basta limpar até ele.

Dois exemplos para esclarecer:

PRIMEIRO:

{ABC}-{BC}+{D}

(**aux1 = ABC e aux2 = BC**)

{**A**BC}-{BC}+{D}

(adiciono por cima da string original, a partir da posição após o { antes o “-”, tudo que está em aux1 mas não está em aux2, o que seria somente “A”)

{A**}**C}-{BC}+{D}

(coloco um **“ } “** após A para **finalizar o conjunto da diferença**)

Como houve diferença entre aux1 e aux2, o conjunto diferença deverá ser menor que o próprio aux1 e dessa maneira limparei tudo até que encontre o segundo **“ } “**, fazendo com que reste:

**{A} +{D}**

SEGUNDO:

{ABC}-{DE}+{C}

(**aux1 = ABC e aux2 = DE**)

{**ABC**}-{DE}+{C}

(adiciono, por cima da string original e a partir do espaço após o **“{“,** tudo que está em aux1 mas não está em aux2; como ambos são completamente diferentes, acabo adicionando aux1 inteira e nada muda)

{ABC**}**-{DE}+{C}

(coloco o **“ } “** ao fim do conjunto diferença, o que também não muda nada)

Agora observe que, partindo da posição após o **“}”** que finaliza a diferença, eu devo limpar tudo até o **primeiro “ } “,** caso contrário acabaria apagando o **+{C}**, que não tem nada haver com isso (daí a importância da *flag* que mencionei, nesse caso ela valeria **0,** pois em nenhum momento houve diferença entre aux1 e aux2, e isso informaria ao programa que ele deve limpar tudo até o primeiro **”}”).**

Portanto sobraria **{ABC} +{C}.**

Observe agora que se eu aplicar sobre tal resultado o algoritmo de resolução de uma soma ele funcionaria perfeitamente, pois ele simplesmente se preocuparia com a posição do último **“ { “,** do **“ + ”** e do **“ { “** subjacente, ignorando completamente esses espaços vazios. A lógica inteira desse código consiste no fato de que ele ignorará completamente espaços vazios.

A intersecção também funciona de maneira semelhante, conhecidas as posições do último **“ { “** , do **“ \* ”**, e do **“ { “** subjacente, encontro aux1 e aux2 e daí passo para a operação propriamente dita.

Basta analisar se o o elemento de aux1 pertence a aux2, se sim então adiciono ele na posição após o primeiro **“ { “** (assim como na diferença), se não então ignoro-o. Mais uma vez é possível que o conjunto intersecção seja menor que aux1, então devo criar uma flag que se indique se devo limpar um ou dois **“}”.** Nesse caso precisarei limpar dois se, em algum momento, existir elemento de aux1 que não se encontra em aux2 (ou seja, aux1 e aux2 são diferentes).

Assim, seguem dois exemplos do funcionamento dessa intersecção:

PRIMEIRO=

{ABC}\*{C}-{ABC}

(**aux1 = ABC e aux2 = C**)

{**C**BC}\*{C}-{ABC}

(somente **“C”** se encontra em ambos ao mesmo tempo, então adiciná-lo-ei ao espaço após o primeor **“{“**)

{C**}**C}\*{C}-{ABC}

(agora coloco o **“}”** responsável por “fechar” o conjunto intersecção. Como houve diferença entre aux1 e aux2 eu devo limpar tudo após o **“}”** até o segundo **“}”** )

O resultado seria **{C} -{ABC}.**

SEGUNDO=

{ABC}\*{ABC}-{DE}

(**aux1 = ABC e aux2 = ABC**)

{**ABC**}\*{ABC}-{DE}

(como ambos são iguais, a adição de aux1 após o primeiro **“{“** nada muda)

{ABC**}**\*{ABC}-{DE}

(adicionar o **“}”** tapando o conjunto intersecção também não muda nada. Como não ocorreu nenhuma diferença entre aux1 e aux2, a flag deve estar valendo 0, o que fará com que o programa apague tudo até o primeiro **“}”** somente)

Ficando assim com **{ABC} -{DE}**

Veja um exemplo de como ele funcionaria numa expressão maior:

{ABC}-{BEF}\*{BDG}+{AG} (começa pelo **“ \* “. aux1 = BEF e aux2 = BDG**)

{ABC}-{**B}**F}\*{BDG}+{AG} (após a colocação da intersecção e do  **“ } “** que a fecha)

{ABC}-{B} +{AG} (após limpeza, que seria até o segundo  **“ } ”**)

{ABC}-{B} +{AG} (não existe mais **“ \* “.** Então vai para **“ - “. aux1 = ABC**

**aux2 = B**)

{**B}**C}-{B} +{AG} (após a colocação da diferença e do **“ } “** que a fecha)

{B} +{AG} (após limpeza, que também vai até o segundo **“ } “**)

{B} +{AG} (agora temos **“ + “.** **aux1 = B e aux2 = AG**)

{B**AG}**  +{AG} (após adição da União e do **“ } “** que a fecha)

{B**AG}** (após limpeza, dessa vez somente até o próximo **“ } “**)

Existem somente outros dois problemas no resultado final. O primeiro são espaços no início, sendo resultado de apagar os parênteses. Por exemplo, em ({ABC}+{AB})\*{A},

seguindo o algoritmo de resolver parenteses, teria:

({ABC}+{AB})\*{A} (realiza a linha dentro do parêntese)

({ABC} )\*{A} (agora apagarei os parênteses)

{ABC} \*{A} (observe que existe um espaço teimoso no ínicio)

{A} (E ele permanece no final, o levaria a um erro de formatação)

Para resolver isso eu posso ultilizar algo semelhante com o que fiz na Revisão de Contrato; eu varro o vetor final com um **for** que me indique a posição a partir da qual existe um elemento diferente de um espaço vazio, e daí torno essa posição o início da string que vou exibir na tela.

O segundo detalhe é simples, preciso exibir o resultado em ordem alfabética. Para fazer isso eu posso ultilizar os conhecimentos que usei em Elfo das Trevas, lembrando que cada caractere é um número na tabela ASCII e usando o algoritmo de ordenação mencionado nessa questão é bem simples organizar o resultado final em ordem alfabética

Dito isso, hora de ver como o código faz tudo isso. (Ele ficou um tanto longo)

**4) Código-Fonte:**

**#include <stdio.h>**

**#include <string.h>**

**char \*vet, aux1[256], aux2[256], \*pi3, \*Inicio, \*Fim;**

**void Save(char \*pi, char \*pi2)**

**{**

**/\* Salvando em aux1 e aux2 os conjuntos que serão operados \*/**

**int i;**

**for (i = 0; \*(pi + 1 + i) != '}'; i++) aux1[i] = \*(pi + 1 + i);**

**aux1[i] = '\0';**

**/\*printf ("AUX1 = %s\n", aux1);\*/**

**for (i = 0; \*(pi2 + 1 + i) != '}'; i++) aux2[i] = \*(pi2 + 1 + i);**

**aux2[i] = '\0';**

**/\*printf ("AUX2 = %s\n", aux2);\*/**

**return;**

**}**

**void Uniao(char \*pi, char \*pi2)**

**{**

**int i, j , g = 1, c;**

**Save (pi, pi2);**

**/\* Fazendo a União propriamente dita \*/**

**i = strlen(aux1);**

**for (j = 0; aux2[j] != '\0'; j++, g = 1){**

**for (c = 0; aux1[c] != '\0'; c++)**

**if (aux2[j] == aux1[c]){**

**g = 0;**

**break;**

**}**

**if (g){**

**\*(pi + 1 + i) = aux2[j];**

**i++;**

**}**

**}**

**\*(pi + 1 + i) = '}';**

**pi3 = pi + 2 + i;**

**for (i = 0; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

**/\*printf ("SOMA FEITA = %s\n", vet);\*/**

**return;**

**}**

**void Dif(char \*pi, char \*pi2)**

**{**

**int i, j, a = 1, c, g = 0;**

**Save (pi, pi2);**

**/\* Fazendo a Diferença propriamente dita \*/**

**for (i = 0, c = 0; aux1[i] != '\0'; i++, a = 1){**

**for (j = 0; aux2[j] != '\0'; j++)**

**if (aux2[j] == aux1[i]){**

**a = 0;**

**g = 1;**

**break;**

**}**

**if (a) {**

**\*(pi + 1 + c) = aux1[i];**

**c++;**

**}**

**}**

**\*(pi + 1 + c) = '}';**

**pi3 = pi + 2 + c;**

**i = 0;**

**if (g){**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i++) = ' ';**

**}**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

**/\*printf ("DIFERENCA FEITA = %s\n", vet);\*/**

**return;**

**}**

**void Inter(char \*pi, char \*pi2)**

**{**

**int i, j, c, g = 0, f = 0;**

**Save (pi, pi2);**

**/\* Fazendo a Intersecção propriamente dita \*/**

**for (i = 0, c = 0; aux1[i] != '\0'; i++, g = 0){**

**for (j = 0; aux2[j] != '\0'; j++)**

**if (aux1[i] == aux2[j]) {**

**g = 1;**

**break;**

**}**

**if (g) {**

**\*(pi + 1 + c) = aux1[i];**

**c++;**

**}**

**else f = 1;**

**}**

**\*(pi + 1 + c) = '}';**

**pi3 = pi + 2 + c;**

**i = 0;**

**if (f){**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i++) = ' ';**

**}**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

**/\*printf ("INTERSECAO FEITA = %s\n", vet);\*/**

**return;**

**}**

**void ResolvedorDeLinha (void)**

**{**

**int i, c, aux;**

**char \*pi, \*pi2, aux2;**

**vet = Inicio;**

**if (\*(Inicio) == '(') vet = Inicio + 1;**

**\*(Fim) = '\0';**

**/\* Primeira Fase: Busca e calculo de Intersecções \*/**

**do{**

**for (i = 0, aux = 0; i < Fim - Inicio; i++){**

**if (!aux && \*(vet + i) == '{') pi = vet + i;**

**if (\*(vet + i) == '\*') aux = 1;**

**if (aux && \*(vet + i) == '{') {**

**pi2 = vet + i;**

**break;**

**}**

**}**

**if (aux) Inter(pi, pi2);**

**} while (aux != 0);**

**/\* Segunda Fase: Busca e Calculo de Operadores gerais (+ e -) \*/**

**do{**

**for (i = 0, aux2 = '/'; i < Fim - Inicio; i++){**

**if (aux2 == '/' && \*(vet + i) == '{') pi = vet + i;**

**if (\*(vet + i) == '+' || \*(vet + i) == '-') aux2 = \*(vet + i);**

**if (aux2 != '/' && \*(vet + i) == '{') {**

**pi2 = vet + i;**

**break;**

**}**

**}**

**if (aux2 == '+') Uniao(pi, pi2);**

**if (aux2 == '-') Dif(pi, pi2);**

**} while (aux2 != '/');**

**}**

**main ()**

**{**

**int i = 0, j, g = 0;**

**char exp[256], aux;**

**while (scanf ("%256s", exp)){**

**Inicio = exp;**

**/\* printf ("LINHA = %s\n", vet); \*/**

**/\* Localizando o parentese mais interno \*/**

**do{**

**if (exp[i] == '(')**

**Inicio = exp + i;**

**if (exp[i] == ')'){**

**Fim = exp + i;**

**g = 1;**

**}**

**i++;**

**if (g){**

**ResolvedorDeLinha();**

**g = 0;**

**i = 0;**

**\*(Inicio) = ' ';**

**\*(Fim) = ' ';**

**}**

**}while (exp[i] != '\0');**

**Inicio = exp;**

**Fim = exp + i;**

**ResolvedorDeLinha();**

**/\* Limpa espaços desnecessarios no inicio \*/**

**for (i = 0; vet[i] == ' '; )**

**vet = vet + ++i;**

**/\* Organizando o vetor inserido em ordem alfábetica \*/**

**for (i = 2; vet[i] != '}'; i++)**

**for (j = 1; j < i; j++)**

**if (vet[j] > vet[i]){**

**aux = vet[j];**

**vet[j] = vet[i];**

**vet[i] = aux;**

**}**

**printf ("%s\n", vet);**

**}**

**}**

**5) Explicação do Código-Fonte:**

Vamos lá. O código é divido em vários módulos, vou começar explicando como as operações são feitas, depois o ResolvedorDeLinha e então a função main (uma explicação “de dentro para fora”).

Começo o programa adicionando os arquivos header stdio.h e string.h, através da diretiva de pré-processamento #include; stdio.h é fundamental para ler e exibir resultados, e string.h eu coloquei somente para usar a função strlen, que eu poderia ter feito na mão facilmente, mas o código já tem módulos até demais.

Em seguida eu defino uma série de variáveis globais:

**char \*vet, aux1[256], aux2[256], \*pi3, \*Inicio, \*Fim;**

**vet** é um ponteiro para caracteres que eu usarei para rececer uma string sem parênteses. **aux1 e aux2** são os auxiliares antes mencionados, coloquei o tamanho igual a 256 pela simplicidade (o tamanho máximo da entrada é 256 caracteres). **pi3** é usado nas limpezas (significa ponteiro inicial 3 – pois marca o início da posição que deve ser limpa – ; daqui a pouco apareceram os 1 e 2). **Inicio** e **Fim** marcam o início da string dentro dos parênteses e o final dela.

Vamos começar falando da função **Save.** Como é bem recorrente a necessidade de colocar os conjuntos antes e depois de um operador em aux1 e aux2, eu criei uma função que faz isso. Ela é bem simples:

**void Save(char \*pi, char \*pi2)**

**{**

**/\* Salvando em aux1 e aux2 os conjuntos que serão operados \*/**

**int i;**

**for (i = 0; \*(pi + 1 + i) != '}'; i++) aux1[i] = \*(pi + 1 + i);**

**aux1[i] = '\0';**

**/\*printf ("AUX1 = %s\n", aux1);\*/**

**for (i = 0; \*(pi2 + 1 + i) != '}'; i++) aux2[i] = \*(pi2 + 1 + i);**

**aux2[i] = '\0';**

**/\*printf ("AUX2 = %s\n", aux2);\*/**

**return;**

**}**

A função não precisa gerar retorno, pois manipula as váriaveis globais **aux1 e aux2**. Ela recebe dois ponteiros, **pi e pi2** (olha os ponteiros iniciais 1 e 2 ai), que são, respectivamente, a posição do último **“ { “** antes do operador e a posição do primeiro **“ { “** após ele.

Dentro da função eu declaro a variável local inteira **i**, o clássico contador para todos momentos. No primeiro **for** eu coloco o conteúdo do primeiro conjunto (antes do operador) em **aux1**; para fazer isso basta colocar o conteúdo de **pi + 1 + i** em **aux1** até que esse conteúdo seja **“ } “**, marcando o final do primeiro conjunto.

Observe que eu adiciono **“ 1 “** ao ponteiro **pi**, para que **i = 0** se refira ao elemento imediatamente após o **“ { “** e não ao **“ { “** propriamente dito.

Após finalizar esse **for** é importante “tampar” **aux1** com um **“\0”** na posição final, para que eu possa lidar com ele como sendo uma string (da mesma maneira que foi feito e detalhado em Revisão de Contrato).

Faço algo idêntico para **aux2**, só que usando **pi2** ao invés de **pi.**

(Os printf’s em comentários ali foram usados por mim durante a resolução para verificar se essas linhas de código estavam funcionando. Se removidos os comentários, é possível observar exatamente quais são aux1 e aux2 em cada etapa do programa)

Finalizado o “Save”, a função retorna ao código original.

Vamos agora entender como funcionam as operações. Cada qual é realizada por uma função, como veremos.

Vamos começar pela mais simples, a União:

**void Uniao(char \*pi, char \*pi2)**

**{**

**int i, j , g = 1, c;**

**Save (pi, pi2);**

**/\* Fazendo a União propriamente dita \*/**

**i = strlen(aux1);**

**for (j = 0; aux2[j] != '\0'; j++, g = 1){**

**for (c = 0; aux1[c] != '\0'; c++)**

**if (aux2[j] == aux1[c]){**

**g = 0;**

**break;**

**}**

**if (g){**

**\*(pi + 1 + i) = aux2[j];**

**i++;**

**}**

**}**

**\*(pi + 1 + i) = '}';**

**pi3 = pi + 2 + i;**

**for (i = 0; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

**/\*printf ("SOMA FEITA = %s\n", vet);\*/**

**return;**

**}**

A função recebe **pi e pi2**, para poder salvar o primeiro e segundo conjunto em **aux1 e aux2** através de Save. Também retorna void, pois modifica o conteúdo de **vet,** que é uma variável global.

Algumas variáveis são declaradas: **int i, j , g = 1, c . i, j e c** são contadores, enquanto **g** é a flag que informará se o elemento de aux1 já se encontra em aux2 ou não.

Em seguida a função Save é invocada, e **pi e pi2** são enviados para ela, garantindo que **aux1 e aux2** recebam os conjuntos que estão antes e após o operador.

Em seguida em atribuo a **i** o tamanho de aux1. Usarei o contador **i** para colocar o conjunto união por cima da string original (observe que ele aponta para o final do primeiro conjunto, posição onde se encontra o **“ } “**), conforme explicado no tópico 3.

**i = strlen(aux1);**

Dai eu crio um ciclo **for**, responsável por mover o contador **j** e através dele eu acesso os elementos de aux2. Para cada elemento dele eu realizo uma comparação com todos os elementos de aux1, através do **for** encadeado e usando o contador **c** para percorrer o aux1.

**for (j = 0; aux2[j] != '\0'; j++, g = 1){**

**for (c = 0; aux1[c] != '\0'; c++)**

**if (aux2[j] == aux1[c]){**

**g = 0;**

**break;**

**}**

**if (g){**

**\*(pi + 1 + i) = aux2[j];**

**i++;**

**}**

**}**

Assim cada aux2[j] será comparado com todos os elementos de aux1. Se em algum momento houver uma igualdade, a flag **g** receberá o valor 0 e o ciclo será finalizado. O desvio **if** subjacente é responsável por determinar se o elemento **aux[j]** deve ser colocado ou não na posição **i (**aquela no final do primeiro conjunto, exatamente onde se encontra o **“ } “**). Se a flag **g** valer 0, é porque em algum momento **aux2[j] foi igual a um elemento de aux1** e portanto ele **não deve ser adicionado**. Já se isso não ocorreu, **g** permaneceu inalterada, valendo 1 e permitindo que aux2[j] seja colocada na posição **i**.

Observe que, se aux2[j] for colocado na posição ao final de aux1, então **i** é acrescido de uma unidade, que será onde o próximo aux2[j] não presente em aux1 será adicionado. Caso não haja outro, a posição **i** será aquela ao final do conjunto união, onde eu deverei colocar a **“ } “** para fechar tal conjunto. (é exatamente isso que a linha **\*(pi + 1 + i) = '}'** faz; perceba também que ele ocorre logo após o fim do ciclo **for** responsável por “correr” aux2).

É importante notar que cada vez que eu passo de um elemento aux2[j] para outro eu atribuo 1 a flag **g** novamente, isso é importante pois cada elemento aux2 pode ou não estar em aux1.

Essas linhas então colocaram o conjunto união por cima da string original, a partir da posição final do primeiro conjunto, além de “tampa-lo” com a **“ } “**. Agora preciso limpar tudo depois de **“ } “** até o próximo **“ } “**, incluindo ele.

Para isso eu noto que **pi + 1 + i**, ao final do posicionamento, é a posição onde está o **“ } “**. Portanto preciso limpar a partir de **pi + 1 + i + 1 ou pi + 2 + i.** Esse valor é a posição inicial de limpeza, que atribuirei à variável **pi3.**

**pi3 = pi + 2 + i;**

O ciclo for seguinte move o contador **i** até que eu encontre um **“ } “**. Essa condição finaliza o ciclo e permite que tudo até tal chave seja apagado, mas o **“ } “** não. Eu o apago após o fim do ciclo com a linha **\*(pi3 + i) = ' '.**

**for (i = 0; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

Dessa maneira fica claro que a função União realiza exatamente o descrito no tópico 3. Vamos ver agora como a função Diferença faz isso:

**void Dif(char \*pi, char \*pi2)**

**{**

**int i, j, a = 1, c, g = 0;**

**Save (pi, pi2);**

**/\* Fazendo a Diferença propriamente dita \*/**

**for (i = 0, c = 0; aux1[i] != '\0'; i++, a = 1){**

**for (j = 0; aux2[j] != '\0'; j++)**

**if (aux2[j] == aux1[i]){**

**a = 0;**

**g = 1;**

**break;**

**}**

**if (a) {**

**\*(pi + 1 + c) = aux1[i];**

**c++;**

**}**

**}**

**\*(pi + 1 + c) = '}';**

**pi3 = pi + 2 + c;**

**i = 0;**

**if (g){**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i++) = ' ';**

**}**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

**/\*printf ("DIFERENCA FEITA = %s\n", vet);\*/**

**return;**

**}**

Aqui eu uso os mesmos contadores **i, j e c**. Só que dessa vez tenho duas flags: **a** e **g**. A primeira delas irá ser usada para indicar se o elemento do primeiro conjunto está presente no segundo conjunto, e assim será efetuada a diferença. Já a segunda indicará se houve uma diferença em algum momento (inclusive mencionei essa no tópico 3), caso tenha havido então terei que limpar os próximos dois **“ } “**, se não terei que limpar somente um **“ } “.**

A função também recebe **pi e pi2**, e usa a função Save para colocar o primeiro e segundo conjunto em aux1 e aux2 como já mencionado.

**for (i = 0, c = 0; aux1[i] != '\0'; i++, a = 1){**

**for (j = 0; aux2[j] != '\0'; j++)**

**if (aux2[j] == aux1[i]){**

**a = 0;**

**g = 1;**

**break;**

**}**

**if (a) {**

**\*(pi + 1 + c) = aux1[i];**

**c++;**

**}**

}

O primeiro ciclo for move i, e através dele eu acesso os elementos de aux1. Para cada elemento eu o comparo com todos aqueles em aux2, através do for encadeado acesso os elementos de aux2 e através do If faço a comparação. Se o elemento de aux1 não estiver em aux2, então a flag **a** se mantém valendo 1, e o desvio if fora do **for** que percorre aux2 permitirá o acesso as linhas de código que sobrescrevem a string original, colocando os elementos apropriados a partir da posição após o primeiro **“ { “.**

Note que essa é a posição **pi + 1 + c** (o +1 faz com que eu comece a contar após o **“ { “**), onde **c** é um contador independente que inicialmente vale 0, mas cada vez que um elemento do conjunto diferença é colocado sobre a string original o **c** é acrescido de uma unidade, fazendo com que o próximo elemento do conjunto diferença entre no próximo espaço na string original e assim por diante.

Observe que cada vez que eu movo de um elemento aux1[i] para outro eu volto a colocar o valor 1 na flag **a**, pois o próximo aux1[i] podem estar em aux2. Note também que quando o **if** subordinado a ambos os ciclos **for** é acessado a flag **g** recebe o valor 1, o que significará que, na hora de limpar os elementos após o conjunto diferença, terei que limpar até dois **“ }”.**

Uma vez finalizado o primeiro ciclo **for** eu preciso “tampar” o conjunto diferença, colocando um **“ } “** ao final dele. Isso é feito através da linha de código **\*(pi + 1 + c) = '}'.**

Mais uma vez a posição a partir da qual eu preciso limpar é **pi + 2 + c,** então atribuo à **pi3** esse valor. Dessa vez existem dois ciclos, ambos limpam de uma certa posição até o encontrar um **“ } “**, onde o ciclo é finalizado e esse **“ } “** também é limpo. Um desses ciclos só é acessado se a flag **g** for não nula, o que faz com que o programa saiba quando deve limpar um e quando deve limpar dois **“ }”.** Observe:

**if (g){**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i++) = ' ';**

**}**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

**/\*printf ("DIFERENCA FEITA = %s\n", vet);\*/**

**return;**

Observe que i = 0 inicialmente, e se os dois ciclos ocorreram, o primeiro “deixa” i com um certo valor que é retomado pelo segundo ciclo, ou seja, o segundo ciclo começa onde o primeiro termina. Se o segundo não ocorrer, então já foi limpo o que era necessário e o código é finalizado.

Nas três operações existe um printf colocado em comentário. Esse mostra exatamente a forma como a string se encontra após passar por tal operação. Assim se os “/\*” foram removidos, é possível ver o passo a passo que o código realiza ao resolver um código, no mesmo estilo daqueles que apresentei no tópico 3.

Vamos agora para a intersecção:

**void Inter(char \*pi, char \*pi2)**

**{**

**int i, j, c, g = 0, f = 0;**

**Save (pi, pi2);**

**/\* Fazendo a Intersecção propriamente dita \*/**

**for (i = 0, c = 0; aux1[i] != '\0'; i++, g = 0){**

**for (j = 0; aux2[j] != '\0'; j++)**

**if (aux1[i] == aux2[j]) {**

**g = 1;**

**break;**

**}**

**if (g) {**

**\*(pi + 1 + c) = aux1[i];**

**c++;**

**}**

**else f = 1;**

**}**

**\*(pi + 1 + c) = '}';**

**pi3 = pi + 2 + c;**

**i = 0;**

**if (f){**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i++) = ' ';**

**}**

**for (i = i; \*(pi3 + i) != '}'; i++)**

**\*(pi3 + i) = ' ';**

**\*(pi3 + i) = ' ';**

**/\*printf ("INTERSECAO FEITA = %s\n", vet);\*/**

**return;**

}

A lógica aqui é altamente semelhante à da diferença; **i, j e c** são contadores, **f e g** são flags. A função de **f** é dizer ocorreu alguma semelhança entre aux1 e aux2, e assim devem existir dois **“ } “** para serem limpos, enquanto **g** irá dizer se aux1[i] foi igual a algum elemento de aux2, se sim então devo colocar esse elemento sobre a string original, a partir da posição após o **“ { “** (início do primeiro conjunto / posição **pi + 1)**.

Imediatamente após as declarações de variáveis é ativada a função Save, que coloca o conteúdo dos conjuntos antes e após o operador em **axu1 e aux2.** O primeiro ciclo **for** move o contador **i,** e através dele eu acesso aux1[i]. Cada aux1[i] será comparado a cada elemento de aux2, através de outro ciclo **for** subordinado ao primeiro e que move **j,** com o qual eu acesso cada elemento de aux2. Como o conjunto intersecção é formado pelos elementos que estão em ambos os conjuntos ao mesmo tempo, eu uso o desvio **if** para verificar se aux1[i] = aux2[j].

Se essa igualdade ocorrer em algum momento a flag recebe o valor 1 e um break força o programa a sair do ciclo de aux2 (isso também acontecia nos outros operadores). Em seguida é validada a flag **g**, se ela valer 1 então o aux1[i] é colocado na posição **pi + 1 + c,** se não então existe elemento de aux1 que não está em aux2, o que leva a necessidade de limpar os próximos dois **“ } “** ao final dese ciclo, assim a flag **f** recebe o valor 1. O ciclo **for** então move o contador **i**, permitindo a análise do próximo elemento de aux1 (e isso se mantém até que aux1 termine).

Observe que a posição onde eu insiro o conjunto intersecção é regida pelo contador **c**, que é recebe o valor zero na primeira vez que o ciclo que move aux1 é acessado. Note também que toda vez que aux1 é colocado por cima da string original o valor **c**, permitindo que a próxima posição seja sobreposta agora, se necessário.

Ao final da analise de aux1 é colocada a chave que fecha o conjunto intersecção através da linha de código **\*(pi + 1 + c) = '}'**, que deve ficar na posição **pi + 1 + c** (pois **c** sempre é acrescido após a posição **pi + 1 + c** receber algum conteúdo, fazendo com que **pi + 1 + c** – já com c aumentado – sempre esteja com um conteúdo desimportante)

A limpeza ocorre de maneira idêntica aquela executada na Diferença, então com perdão da objetividade vou omiti-la.

Apesar de longos, esses trechos são básicos e executam precisamente a ideia presente no tópico 3. Uma vez entendendo como esses operadores funcionam, vamos agora ver como a função ResolvedorDeLinha funciona.

**void ResolvedorDeLinha (void)**

**{**

**int i, c, aux;**

**char \*pi, \*pi2, aux2;**

**vet = Inicio;**

**if (\*(Inicio) == '(') vet = Inicio + 1;**

**\*(Fim) = '\0';**

**/\* Primeira Fase: Busca e calculo de Intersecções \*/**

**do{**

**for (i = 0, aux = 0; i < Fim - Inicio; i++){**

**if (!aux && \*(vet + i) == '{') pi = vet + i;**

**if (\*(vet + i) == '\*') aux = 1;**

**if (aux && \*(vet + i) == '{') {**

**pi2 = vet + i;**

**break;**

**}**

**}**

**if (aux) Inter(pi, pi2);**

**} while (aux != 0);**

**/\* Segunda Fase: Busca e Calculo de Operadores gerais (+ e -) \*/**

**do{**

**for (i = 0, aux2 = '/'; i < Fim - Inicio; i++){**

**if (aux2 == '/' && \*(vet + i) == '{') pi = vet + i;**

**if (\*(vet + i) == '+' || \*(vet + i) == '-') aux2 = \*(vet + i);**

**if (aux2 != '/' && \*(vet + i) == '{') {**

**pi2 = vet + i;**

**break;**

**}**

**}**

**if (aux2 == '+') Uniao(pi, pi2);**

**if (aux2 == '-') Dif(pi, pi2);**

**} while (aux2 != '/');**

**}**

Eu inicio essa função declarando algumas variáveis:

**int i, c, aux;**

**char \*pi, \*pi2, aux2;**

**i e c** serão os contadores de sempre. **Aux** é um auxiliar do tipo inteiro (não confundir com aux1, string que captura o primeiro conjunto de uma operação), e **aux2** é um auxiliar do tipo caractere (também não deve ser confundido com a string aux2). **Pi e pi2** são os ponteiros já mencionados, que guardam a posição do último **“ { “** antes do operador e do **“ { “** após ele.

O código começa se utilizando das variáveis Início e Fim. Na função main elas são utilizadas para guardar as posições dos **“ ( “** e **“ ) ”** mais internos, e aqui serviram para montar a string **vet** que possui como conteúdo os elementos dentro dos parênteses (observe que essa string não conterá nenhum parêntese, pois já estou lidando com o parêntese mais interno).

**vet = Inicio;**

**if (\*(Inicio) == '(') vet = Inicio + 1;**

**\*(Fim) = '\0';**

No algoritmo que eu construí no tópico 2 eu mencionei a necessidade de executar o resolvedor de linha uma última vez após o fim dos parênteses. Pensando nisso, eu atribuo ao início de **vet** o ponteiro **Inicio,** mas em seguida verifico se o conteúdo de **Inicio** de fato é um parêntese (pois, na última vez que eu precisar executar o ResolvedorDeLinha, ele não será); se for, então eu faço a string **vet** começar na posição após esse parêntese; se não for então **vet** fica onde está mesmo.

Em seguida pego a posição **Fim** e coloco um “ \0 ”,dessa forma posso tratar **vet** de fato como uma string cujo conteúdo era aquilo entre os parênteses.

**/\* Primeira Fase: Busca e calculo de Intersecções \*/**

**do{**

**for (i = 0, aux = 0; i < Fim - Inicio; i++){**

**if (!aux && \*(vet + i) == '{') pi = vet + i;**

**if (\*(vet + i) == '\*') aux = 1;**

**if (aux && \*(vet + i) == '{') {**

**pi2 = vet + i;**

**break;**

**}**

**}**

**if (aux) Inter(pi, pi2);**

**} while (aux != 0);**

Como o comentáro já menciona, nessa parte eu procuro e realizo intersecções. Para isso eu vou me valer da função **Inter** já explicada. Mas antes de ir para ela eu preciso obter **pi e pi2,** posições do **“ { “** antes do **“\*”** e do **“ { “** após esse operador.

Para isso me valho de um ciclo **do while,** responsável por procurar intersecções e continuar procurando até que não haja mais nenhuma. No caso ele vai parar de repetir quando **aux** for não nulo, o que ocorrerá se nenhum **“\*”** for encontrado. Dessa maneira **aux** acaba funcionando como uma flag.

O ciclo **for** por sua vez é responsável por varrer a string **vet** em busca de **“\*”.** Nela é feito um ciclo onde toda vez que um “ { “ aparece sua posição é guardada em **pi**, no entanto isso se mantém somente enquanto nenhum “ \* “ for encontrado, pois quando isso ocorrer **aux** receberá o valor 1, “travando” o primeiro **if** (que era responsável por guardar a posição dos “ { “ que encontrasse). Agora o for continua, mas buscando pelo próximo “ { “; sendo que imediatamente após ele ser encontrado sua posição é guardada em **pi2** e o ciclo for é interrompido pelo comando **break.**

(A lógica aqui é que existem “n” chaves “ { “ até encontrar o operador “ \* “. Eu simplesmente guardo a posição de todas que encontro, de modo que se eu já havia guardado uma posição e encontro outra, simplesmente a sobreponho. Assim quando encontrar um “ \* “ terei, em **pi,** a posição da última “ { “. A “ { “ mais próxima após “ \* “ é exatamente o que eu desejo guardar em **pi2**, então quando eu encontrá-la coloco sua posição em **pi2** e finalizo a busca)

Observe que a váriavel aux, responsável por “fechar” o primeiro **if** e “abrir” o segundo, é usada também para saber se existe ou não “ \* “ em **vet.** Através de **if (aux) Inter(pi, pi2)** eu verifico se devo ou não realizar uma intersecção (atra´ves da função Inter) com base no valor de aux. Esse valor também é o responsável por finalizar o ciclo **while** de busca às **“\*”,** como mencionado.

(Observe que, no for, o contador **i** é limitado por **Fim – Inicio**; essa diferença entre ponteiros é exatamente o tamanho de **vet**, por isso foi usada aqui)

Uma vez que todas as “ \* “ tenham sido resolvidas, passo a me preocupar com “+” e “-”.

**/\* Segunda Fase: Busca e Calculo de Operadores gerais (+ e -) \*/**

**do{**

**for (i = 0, aux2 = '/'; i < Fim - Inicio; i++){**

**if (aux2 == '/' && \*(vet + i) == '{') pi = vet + i;**

**if (\*(vet + i) == '+' || \*(vet + i) == '-') aux2 = \*(vet + i);**

**if (aux2 != '/' && \*(vet + i) == '{') {**

**pi2 = vet + i;**

**break;**

**}**

**}**

**if (aux2 == '+') Uniao(pi, pi2);**

**if (aux2 == '-') Dif(pi, pi2);**

**} while (aux2 != '/');**

Aqui eu realizo algo similar, só que dessa vez o auxiliar é **aux2,** que foi definido como caractere pois eu gostaria que ele, além de “abrir” e “fechar” os **if’s** no **for** que busca pelas posições de “ { “ e “ { “, também me dissesse se eu deveria realizar uma União (através da função Uniao antes explicada) ou uma Diferença (através da função Dif antes mencionada).

Assim o algoritmo se torna praticamente igual. O **Do while** é usado paras manter a busca enquanto houver “+” ou “-” em **vet,** sendo regulado pelo valor de **aux2,** que é tem “ / “ nativamente, mas que sofre uma modificação e passa a acomodar “ + “ ou “ - “ caso um desses seja localizado na string.

Através do **for** é varrida a string **vet** em busca da posição **pi e pi2**. Aqui a lógica é a mesma, vou guardando a posição de todos os “ { “ que eu achar em **pi** até encontrar um “ + “ ou “ - “, daí aux2 recebe esse caractere, “travando” o primeiro if e “desbloqueando” o segundo. Na ultilização do segundo **if,** quando um “ { “ for encontrado guardo sua posição em **pi2** e saio do ciclo **for** de busca.

Agora dois if’s alinhados vão analisar o conteúdo de aux2; se for um “ + “, a função Uniao é executada; se for um “ - “, a função Dif é executada; se for um “ / “, isto é, em nenhum momento houve um “+” ou “-”, nada ocorre e a validação do while seria responsável por colocar um fim ao ciclo de busca por esses operadores.

Com isso conclui-se a explicação da função ResolvedorDeLinha, agora basta garantir que chegue nela as linhas dentro de parênteses. Vamos ver como a função main faz isso:

**main ()**

**{**

**int i = 0, j, g = 0;**

**char exp[256], aux;**

**while (scanf ("%256s", exp)){**

**Inicio = exp;**

**/\* printf ("LINHA = %s\n", vet); \*/**

**/\* Localizando o parentese mais interno \*/**

**do{**

**if (exp[i] == '(')**

**Inicio = exp + i;**

**if (exp[i] == ')'){**

**Fim = exp + i;**

**g = 1;**

**}**

**i++;**

**if (g){**

**ResolvedorDeLinha();**

**g = 0;**

**i = 0;**

**\*(Inicio) = ' ';**

**\*(Fim) = ' ';**

**}**

**}while (exp[i] != '\0');**

**Inicio = exp;**

**Fim = exp + i;**

**ResolvedorDeLinha();**

**/\* Limpa espaços desnecessarios no inicio \*/**

**for (i = 0; vet[i] == ' ';)**

**vet = vet + ++i;**

**/\* Organizando o vetor inserido em ordem alfábetica \*/**

**for (i = 2; vet[i] != '}'; i++)**

**for (j = 1; j < i; j++)**

**if (vet[j] > vet[i]){**

**aux = vet[j];**

**vet[j] = vet[i];**

**vet[i] = aux;**

**}**

**printf ("%s\n", vet);**

**}**

**}**

Eu inicio a função main declarando algumas váriaveis:

**int i = 0, j, g = 0;**

**char exp[256], aux;**

**i e j** servirão como contadores, g será uma flag, exp[256] é a string que receberá a expressão completa que deverá ser avaliada e aux é um auxiliar do tipo caractere (confundir com aux1) que será importante em certa parte do código.

**while (scanf ("%256s", exp)){**

**(...)**

**}**

Aqui eu uso um scanf para ler a expressão que irá ser avaliada – para evitar um erro de fragmentação eu coloquei um limitador de tamanho para a string inserida, apesar de que isso não é tão necessário no URI, já que o tamanho máximo da entrada já é garantido –.

O ciclo while irá repetir as linhas de código subjacentes enquanto for encontrada alguma entrada. Logo de início eu atribuo à variável Inicio o ponteiro para inicio de exp, que é o próprio exp (pois este é um identificador para a posição inicial dessa string).

(Existe também um comentário onde existe um printf que mostra a linha que será resolvida, se os marcadores de comentário forem removidos é possível observar de perto qual linha o programa está analisando)

**/\* Localizando o parentese mais interno \*/**

**do{**

**if (exp[i] == '(')**

**Inicio = exp + i;**

**if (exp[i] == ')'){**

**Fim = exp + i;**

**g = 1;**

**}**

**i++;**

**if (g){**

**ResolvedorDeLinha();**

**g = 0;**

**i = 0;**

**\*(Inicio) = ' ';**

**\*(Fim) = ' ';**

**}**

**}while (exp[i] != '\0');**

A linhas imediatamente após servem para encontrar a posição do parêntese mais interno. Para isso eu crio um ciclo do while que percorre toda a string exp em busca de “ ( “ e “ ) “. Sempre que uma “ ( “ é encontrada eu guardo sua posição em inicio e continuo o ciclo; no entanto quando encontro o primeiro “ ) “ ativo a flag g, tornando seu valor igual a 1 e permitindo que a função ResolvedorDeLinha seja ativada. Observe que a posição desse “ ) “ é guardada no ponteiro para caracteres Fim. Note também que Fim e Inicio são váriaveis globais, assim mudanças feitas em Inicio e Fim a qualquer momento do código irão interferir nessas váriaveis se elas aparecerem em outras funções (e, como já visto, elas aparecem no ResolvedorDelinha).

Após a “função mágica” ser ativada, eu zero o contador i (que é movido em uma unidade a cada ciclo do while) e anulo também a flag; com isso irei voltar a analisar a expressão inteira inserida a partir do início, com uma flag que só voltará a ativar o if novamente se outro “ ) “ aparecer (e por consequência deve haver um “ ( “ antes também, pois as expressões são sintaticamente corretas)

Note que antes de sair do caso if eu troco o conteúdo das posições Inicio e Fim por um espaço vazio, dessa maneira elimino os parênteses mais internos e evito que eles sejam “contados” novamente.

Eventualmente todos os parênteses estarão resolvidos, e assim o ciclo moverá o contador **i** através da string inteira e não encontrará qualquer “ ) “. Quando o último elemento dela, o “ \0 “, for alcançado, o ciclo se finaliza (observe que enquanto houver algum parênteses na expressão esse final nunca será alcansado, pois antes dele deve haver algum i tal que exp[i] = “ ) “ para a expressão ser sintaticamente correta; e toda vez que um “ ) “ é localizado, o if é ativado e o i é zerado, recomeçando a contagem e impedindo que ele chegasse ao final)

**Inicio = exp;**

**Fim = exp + i;**

**ResolvedorDeLinha();**

Finalizado esse ciclo, terá que ser executada a função ResolutorDeLinha uma última vez, e dessa vez a função vet que ela manipulará deve ser igual a função exp restante, isto é, o inicio de vet (que é guardada pelo ponteiro Inicio) vai ter a posição inicial de exp (por isso Inicio = exp), e a posição final de vet (guardada pelo ponteiro Fim) vai ter a posição final de exp.

Como i não foi mais manipulado e ele foi “deixado” com um valor tal que exp[i] = ‘\0’, temos que exp + i deve ser exatamente a posição final do vetor exp, dai esse deve ser exatamente o valor que Fim vai receber. Uma vez resolvidos os Inicio e Fim, a função ResolvedorDeLinha é executada uma última vez, deixando uma string exp com um único conjunto.

Para finalizar eu devo lidar com os casos mencionados no tópico 3. O primeiro deles se refere aos espaços no início, que devem ser apagados. Para isso eu crio o ciclo for abaixo:

**/\* Limpa espaços desnecessários no inicio \*/**

**for (i = 0; vet[i] == ' ';)**

**vet = vet + ++i;**

Aqui eu uso o ciclo para mover o contador i, zerado de novo, e vou aumentando seu valor enquanto vet[i] for um espaço vazio. Ao mesmo tempo atribuo a vet, posição inicial da string vet, a posição vet + ++i (eu acrescento o i primeiro, pois se a próxima posição não for um espaço vazio quero que o inicio de vet esteja nela). Assim eu modifico o endereço de inicio da string vet para o primeiro elemento diferente de um espaço vazio (que será uma chave).

Agora lido com o segundo caso especial mencionado no final do tópico 3, que é organizar o conjunto em ordem alfábetica:

**/\* Organizando o vetor inserido em ordem alfábetica \*/**

**for (i = 2; vet[i] != '}'; i++)**

**for (j = 1; j < i; j++)**

**if (vet[j] > vet[i]){**

**aux = vet[j];**

**vet[j] = vet[i];**

**vet[i] = aux;**

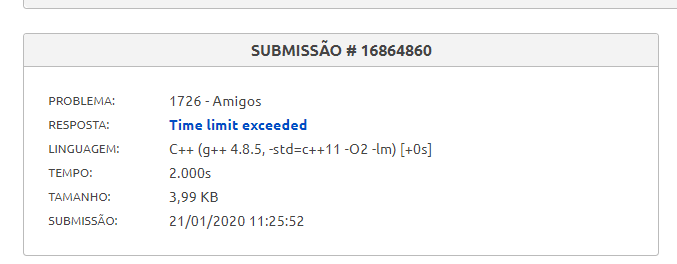
**}**

Aqui eu vou usar o algoritmo de ordenação que usei em Elfo das Trevas, só que devo atentar para não colocar as chaves no meio dessa organização, ou acabarei com uma string errada. Para isso eu coloco i = 2 de inicio (pois, pelo meu algoritmo, eu deveria começar com i sendo o primeiro valor posterior a j. Como j vai ter que valer 1, pois para j = 0 tenho uma chave, i vai ter que ser 2 ) e j igual a 1.

Sigo o algoritmo igual mencionado, comparo cada elemento que eu pegar com todos os anteriores; toda vez que um anterior for maior que o que eu estou usando para comparar eu os troco de posição (pois quero os menores na frente – já que em ASCII as letras do alfabeto estão organizadas em ordem alfabética com as primeiras estando em posições antes das posteriores, como já mencionado –)

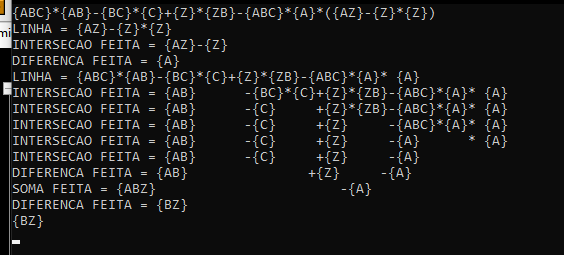
Assim enfim eu tenho o resultado procurado, um conjunto que represente o resultado da expressão inserida, sem espaços iniciais e em ordem alfabética.

**6) Observações finais:**

– A questão apresentada possui todas as saídas condizentes com o exemplo na questão e com os exemplos no uDebug, além de que eu realizei uma série de testes pessoais e ela sempre apresentava a saída correta (as vezes eu errava na solução e ela acertava inclusive), no entanto a questão **excede o limite de tempo de execução permitido.**

No entanto, fui informado de que, como na nossa disciplina não estudamos algoritmos de complexidade e formas otimizar programas, esse erro seria tolerável se a questão funcionasse como esperado.

Caso queira se convencer ainda mais de que ela funciona, basta remover os marcadores de comentário nos comandos de printf que mencionei anteriormente, assim **é possível ver o código resolvendo o problema passo a passo.**

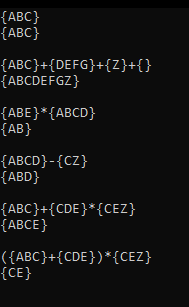
****

“Desbloqueando” todos os comandos “presos” em comentários, é possível ver a linha que o programa vai resolver (que é aquela dentro do parêntese mais interno na expressão), em seguida vê-lo resolvendo-a, através da execução sistemática de cada operação.

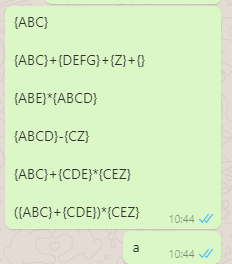
– Note que ao final da main, quando eu passo somente a remover os espaços vazios no início e organizar o conjunto, vet e exp são exatamente iguais. Isso ocorre pois quando eu executo o ResolvedorDeLinha pela última vez (já fora do ciclo de busca por parênteses), o Inicio recebe o ponteiro para inicio de exp e o Fim o ponteiro para o final de exp, e **dentro** do ResolvedorDeLinha vet recebe como seu início o ponteiro Inicio e como final o ponteiro Fim.

Dessa maneira, nas partes posteriores a esse último ResolvedorDeLinha, tanto faz lidar com vet ou exp. Inclusive ultilizei vet, simplesmente por escolha pessoal.

-A questão garante que todas as entradas serão sintáticamente corretas, então eu não me responsabilizo por casos onde a entrada não possui a estrutura esperada.

****

-Um pequeno problema de formatação ocorre quando se copia e cola as entradas presentes no site da questão. (imagem ao lado)

 Eu estava um tanto confuso sobre o que estava acontecendo ali, mas quando conversei com um amigo pelo whatsapp e enviei as entradas para ele, percebi o problema (imagem ao lado)

O que ocorre é que as na página do Uri tem um espaço entre elas (um “\n”), e como esse “\n” não é lido pelo meu código (o scanf (“%256s”, exp) não o lê), ela acaba sendo uma “entrada morta”, isto é, um valor inserido mas que não é interpretado pelo código. Assim o código captura a string válida, apresenta o resultado, em seguida existe um espaço vazio (da entrada) que simplesmente é colocado e ignorado pelo código, depois vem outra string válida, e é apresentado o resultado, ai espaço vazio e assim vai.

De toda forma se as entradas forem inseridas manualmente, ou se forem utilizadas aquelas presentes no uDebug, esse tipo de problema não ocorre (pois não existe esse espaço entre as entradas). (e mesmo com esses espaços as saídas continuam corretas, o programa não tem culpa se colocaram um “\n” ali do nada)

**7) Tópicos abordados na questão:**

**1)Fundamentos de C:** Estrutura geral, Diretivas de Pré-processamento e Função main.

**2)Fundamentos de C:** Dados primitivos, operadores matemáticos e relacionais, funções de entrada e saída de dados (printf e scanf)

**3)Fundamentos de C: Saída** formatada de dados.

**4)Estruturas de Controle de Fluxo:** Desvios (simples, composto, aninhado)

**5)Estruturas de Controle de Fluxo:** Laços (for, while e Do while)

**6)Estruturas de Controle de Fluxo:** Comando Break;

**7)Vetores**: Declaração, Inicialização e Manipulação

**8)Strings:** Declaração, Inicialização e Manipulação

**10)Funções de manipulação de Strings:** strlen.

**11)Ponteiros:** Definição, Declaração, Inicialização, Vetores como ponteiros.

**12)Funções:** Estrutura, Entrada e saida de dados, passagem de parâmetros por valor e por referência.

**13)Escopo de Variáveis:** Globais, Locais, Formais.