Documentação do Trabalho Prático 0

Luísa Ribeiro Bezerra

Abril, 2016

1. Introdução

O objetivo principal deste trabalho é desenvolver um tipo de aplicabilidade da Notação Polonesa Reversa, juntamente também com todo o conhecimento prático adquirido durante todo percurso curricular até aqui, trazendo o envolvimento de uma lógica complexa para resolver tal situação dada. Em tal problema, foi pedido a implementação de um código que receba uma expressão corrompida em Notação Polonesa Reversa. Eu deveria desenvolver um código que retornasse a lista de possíveis sequências de operadores para tal ocasião dada.

Notação Convencional	Notação Polonesa Reversa
$a + b$	$ab+$
$(a + b)/c$	$ab + c/$
$((a + b) * c)/(d - e)$	$ab + c * de - /$

Tabela 1: Exemplos de operações na Notação Polonesa Reversa.

Para desenvolver a lógica do problema, tive que estudar a Notação Polonesa Reversa minuciosamente e entender sua proposta. A notação não é nada mais do que uma pilha de dados, a qual a cada leitura da linha de entrada dada, tomamos uma decisão entre um número na leitura, iremos empilhá-lo, caso entre um operador, iremos retirar os dois últimos valores da pilha e realizar a operação sob eles. Assim sendo, no final, teremos somente um número na pilha, que será o resultado final de todas as operações contidas na linha do caso.

A lógica da Notação Polonesa Reversa não é muito complexa, mas para resolver a situação dada, em que a expressão estava "corrompida" - os operadores * e + apresentavam-se igualmente em formato "?" -, foi desenvolvido um sistema de lógica baseado no número de possibilidades de representação de bits. Implementei a ideia da Máscara de Bits para criar as possíveis combinações de + e * e, através dela, pude criar várias linhas possíveis de entrada, as quais passaram por um sistema de validação para conferir se o seu resultado seria igual ao resultado esperado dado também na entrada e assim, posteriormente, criar a saída desejada.

2. Solução do Problema

2.1. Modelagem

Para a estruturação do problema, criei algumas estruturas para apoio, como a estrutura de uma pilha – bem como a criação de funções de empilhar, desempilhar e criar uma pilha vazia. Foi interessante, neste caso, usar a ideia de pilha para resolver as operações em cima de cada linha que gerei como opção para a resolução do problema. Esta foi a maneira que usei para resolver a Notação Polonesa Reversa. Para a criação desta estrutura, declarei uma estrutura do tipo "nodo", que tem em si mesmo um valor do tipo inteiro e dois ponteiros, um apontando para o próximo nodo da pilha, e o outro para o nodo anterior da pilha. A estrutura da pilha contém dois ponteiros, sendo que um aponta para o nodo topo da pilha, e o outro para o nodo base da pilha.

A ideia de resolução do problema que tive foi baseada no número de possibilidades de representação de bits. Temos, como dado, um valor de "?" de entrada possíveis. Pensei, então, em fazer todas as possibilidades de bits entre todos os "?" dados, ou seja, teria um total de 2^(número de "?") possibilidades a fazer. Tendo isso em mãos, apliquei outro raciocínio juntamente deste problema: para resolver esta , implementei a ideia da Máscara de Bits. Cada opção de possibilidade seria representada como um binário - ou seja, a opção "zero" seria representada com 0000, a opção "um" seria 0001, e assim por diante. Como tenho 2^(número de "?") opções de possibilidades para fazer, tenho todas as opções plausíveis de máscaras de bits para um dado número de "?". Utilizei o próprio número da opção para transformá-la em binário, o que facilitou o desenvolvimento do mesmo, e então tive todas as opções plausíveis. A cada máscara, fiz a criação de uma linha (cópia da linha dada na entrada padrão), porém substituindo os "?" por + (caso a máscara indicasse a leitura de um 0) ou * (caso a máscara indicasse a leitura de um 1). Ela, então, passa por uma validação, ou seja, executo a leitura da linha e transformo-a em uma pilha, fazendo a resolução da Notação Polonesa Reversa dela, e obtendo um resultado. Caso este resultado dê de encontro com o resultado esperado, teria, então, uma possível solução para a questão - e, juntamente, a máscara de bits ideal para ser representada como resposta. Para ter a saída desejada, printei a máscara de bits seguindo uma lógica simples: onde tem 0, imprime +; onde tem 1, imprime *.

2.2. Implementação

O fluxo do programa é iniciado com a leitura da linha dada como entrada padrão para o problema, a qual contém números e "?", sendo todos espaçados por espaços vazios, ou seja, " ". Guardo esta linha lida em uma string, e em seguida leio a segunda entrada dada, que é o número representante da resposta que desejo obter com aqueles possíveis operandos e operadores dados na linha. Há um iterador, que faz uma contagem, dentro da string "linha", do número de "?" existentes nela. Este número é importante, pois ele determina o tamanho da máscara de bits, bem como o número de possibilidades de máscaras existentes, que é de 2^(número de "?"). Com todas essas informações em mãos, há a chamada da

função "possibilidades", que recebe como parâmetros o número de "?", a string linha e a resposta esperada.

Na função "possibilidades", temos então um for, que cria todas as possibilidades dado o número de "?". Para tal, fiz uma iteração usando um inteiro "i" que começa em 0 e percorre todas as possibilidades possíveis. É dentro deste for que basicamente tudo ocorre, uma vez que a cada opção gerada, há a execução de praticamente toda a minha lógica: utilizo uma variável de inteiros chamada "cont", que recebe o número de "?" diminuído de 1 (já que pretendo que ele chegue ao 0 – temos 4 "?", indo de 0 a 3, por exemplo). Crio um while, e enquanto este "cont" for maior ou igual a zero, estarei montando a máscara de bits, baseado em qual possibilidade estou (transformada em binário). Comecei do número máximo em direção ao zero simplesmente porque a lógica da criação de uma máscara, baseada na conversão de um número em binário, é feita pela escrita da direção direita para a esquerda, ou seja, completarei primeiro a última casa, após isso, a penúltima, e assim por diante, até chegar na posição mascaraBits[0] do vetor de inteiros que representa a máscara de bits. Uso o número "i" de possibilidades (guardado sempre na variável "numeroPossibilidades", a fim de não ficar mexendo no "i", já que ele é usado para contagem do número de possibilidades também [do for]) para criar a máscara. Começo pelo 0, ou seja, 0000 (quantidade de zeros relativa à quantidade de "?" dados no problema) é o desejado a se obter. Faço a conta de conversão binária: na última posição do vetor mascaraBits, guardo numeroPossibilidade%2, ando para a casa da esquerda, e nela usarei o valor dado da divisão numeroPossibilidades/2. Esta lógica é seguida até o fim do while, ou seja, até a casa mascaraBits[0] ser escrita, criando a máscara por completo.

Tendo a máscara em mãos, ela será "revelada" na string dada como entrada. Aqui tenho um while, que faz a leitura de toda string e repassa para outra string temporária a cópia da string de entrada. A string temporária vai recebendo o valor da string de entrada e, quando lemos um "?", este "?" será substituído pelo valor de + ou *, baseado na leitura da máscara de bits (agora, lida da esquerda para a direita). A cada vez que encontramos um "?", lemos a máscara, substituímos e andamos uma casa para a direita na máscara de bits, fazendo isso até seu final. A linha, então, é totalmente copiada na linha temporária, porém contendo os + e * baseados na máscara padrão passada. Esta linha temporária, então, é repassada para uma função de validação, chamada "validacao" e com retorno de inteiro – sendo 0 ou 1.

A função validacao recebe linha temporária que criei e o resultado esperado como parâmetros. Para esta parte, resolvi incrementar a lógica da Notação Polonesa Invertida seguindo a incrementação de uma pilha. Crio uma pilha e leio a linha temporária seguindo a utilização do método do strtok para tokenizá-la. Enquanto o token não for vazio, isso é, não tiver chegado ao fim da linha, quando a leitura nos indica um operador, desempilho os últimos dois elementos da pilha, faço a soma ou a multiplicação (baseado no operador lido) e empilho o resultado na pilha. Caso contrário, o token será uma string, e então a transformo em um número inteiro e guardo em minha pilha. Isso será feito até a linha ser totalmente lida e termos somente um número em seu conteúdo, que corresponde ao número final da operação dada na linha temporária. Caso o resultado desta dada possibilidade bata com o resultado esperado dado na entrada padrão, retorno 1 (true). caso contrário, retorno 0 (falso).

Por fim, toda vez que a possibilidade dada tiver um resultado "true" retornado pela validação, irei chamar a função "imprimeSinais", a qual recebe a máscara de bits e o número de "?", ou seja, o tamanho da máscara, para fazer um iterador e imprimir a saida padrão. Enquanto itero nesta máscara, a cada 0, printo +, e a cada 1, printo *. Por fim, o "for" incrementa ao "i" uma unidade, o que nos leva à segunda possibilidade, e teremos o número de i, que é 1, para transformar em binário e criar a máscara, em seguida testá-la, e assim sucessivamente, até o final das possibilidades apresentadas.

3. **Análise de Complexidade**

Nesta seção, será apresentada a análise do custo teórico de tempo e de espaço dos principais algoritmos e estruturas de dados utilizados. Portanto, foram analisadas as funções abaixo contidas na tabela:

validacao	A função tem uma repetição representada pelo while que passa por todos os elementos da entrada. Dentro dele, temos um if/else que, de duas, uma: ou adicionam um elemento na pilha (O(1)), ou realizam a retirada de dois elementos (2*O(1)), e logo após faz um cálculo (O(1)) entre eles e empilha o resultado na pilha (O(1)). Ou seja, a função tem complexidade O(n).
imprimeSinais	A função tem uma repetição representada pelo while que percorre o tamanho da máscara de bits e realiza um if/else que contém apenas printf (O(1)), ou seja, a função tem complexidade O(n).
possibilidade	A função conta com um grande "for", que repete D vezes, onde D = 2^(número de "?"). Dentro deste for, temos um while que repete "n" vezes internamente (O(n)), sendo n = número de "?", e outro while que lê a entrada da linha (O(m)), sendo m = tamanho da entrada. Temos também um if com O(1). Em suma, esta complexidade é 2^(m+n).
criaPilha	A função somente inicializa a pilha com ponteiros nulos, o que traz uma complexidade de O(1).
empilhaElemento	A função insere o elemento na pilha seguindo o seu "topo", ou seja, não há operação nada custosa nesta situação. Como ele somente muda o endereço de onde seus ponteiros apontam, esta função é vista como O(1).
desempilhaElemento	A função retira o elemento do topo da pilha, ou seja, não há operação nada custosa nesta situação. Há a criação de um ponteiro auxiliar, mas ele não interfere na complexidade da situação. Esta função é vista como O(1).

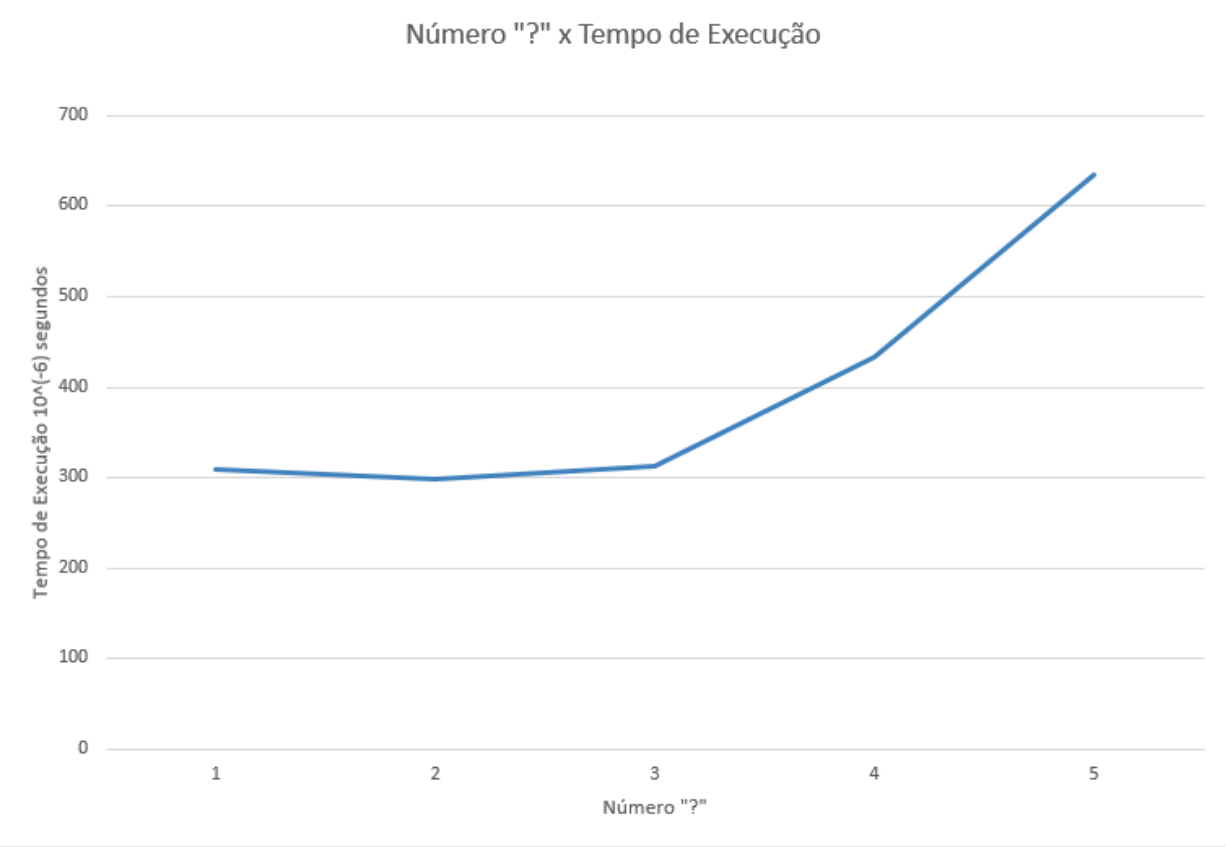
A complexidade total do programa é a maior complexidade dentre as citadas acima.

4. Avaliação Experimental

Para a avaliação experimental, utilizei os testes "toy" cedidos pelo professor, bem como também testes desenvolvidos por mim mesma e colegas da disciplina. Os experimentos foram executados em um sistema com 8 GB de memória RAM e um processador Intel Core i7-3632QM CPU @ 2.20GHz.

4.1. Testes Toy

Para os testes Toy fornecidos pelo professor, criei um gráfico baseado no número de "?" de entrada fornecida versus o tempo de execução. Sempre que repeti o teste para a mesma entrada, o tempo de execução foi bem aproximado, o que mostra que o algoritmo é bem estável. Foram repetidas 10 vezes cada teste e feito a média deles. Segue o gráfico om a média dos tempos de execução:



4.2. **Testes Desenvolvidos**

Para os testes criados por mim e pelos colegas, criei um gráfico baseado no número de "?" de entrada fornecida versus o tempo de execução. Sempre que repeti o teste para a mesma entrada, o tempo de execução também foi bem aproximado, o que mostra que o algoritmo é bem estável. Foram repetidas 10 vezes cada teste e feito a média deles. Segue o gráfico com a média dos tempos de execução:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Série1	295	269	320	343	373	595	676	1160	2966	3919	9380	14883	37816	53331	119766	219120	461715	906630	1.886.0	4.098.8	7873772	1,6E+07	3,5E+07	6,8E+07	1,4E+08	3,1E+08

5. **Conclusão**

Por mais que eu tenha tentado otimizar o código, chegando a resolver este Trabalho Prático com duas soluções possíveis, quanto maior a string de entrada (e com mais "?" consequentemente), maior a máscara de bits, o que ocasiona em maior número de possibilidades. Temos que o número de possibilidades em bits nos dá 2^(número de "?") opções, e quando a entrada é grande, o número de "?" também, ocasionando num aumento exponencial do tempo de execução e tornando o algoritmo bem custoso. Sendo esta a função mais custosa de todo programa principal, ela que determina praticamente todo o custo do mesmo, então acaba sendo realmente custoso a entrada grande.

O trabalho foi fundamental para a aplicabilidade de teorias que aprendi durante o curso, e pude aplicar a ideia de máscara de bits, implementar pilhas para ter um custo menor do algoritmo, pude trabalhar a ideia de custo em geral e tentar otimizar o tempo de execução, bem como a complexidade, do meu trabalho. Achei a proposta muito interessante e pude, com toda clareza, desenvolver minhas habilidades de lógica e programação.