Documentação TP0

Aluno: Semar Augusto da Cunha Mello Martins

Introdução:

O problema a ser resolvido no TP é um problema em que o pior caso é 2ˆn a solução. Ele consiste em calcular todas as possibilidades de combinações de adição e multiplicação entre os números da entrada (mantendo sempre a ordem correta dos números da expressão dada). O maior problema está no fato de que calcular todas as combinações entre 0s e 1s (seja 0 uma adição e 1 uma multiplicação) é extremamente caro para uma string grande. Devido ao número extremamente alto de combinações entre operações, o numero de adições e multiplicações feitas pelo computador também se torna muito alto, já que para cada combinação de operadores serão feitas n-1 (seja n o numero de números na expressão) contas em um programa simples.

Metodologia:

Como os números eram apenas inteiros positivos, usei o valor -1 para identificar a interrogação, para facilitar as contas, já que assim seria possível usar um vetor de inteiros e não um de char. De maneira semelhante, considerei que o operador + seria simbolizado por 0 e o operador * por 1 em um vetor que possuía tantas posições quanto tivessem interrogações na entrada. Pois dessa forma seria possível utilizar de um contador simples para calcular todas as permutações necessárias.

Para a implementação do tp foi usado uma estrutura de pilha padrão e um método usando loops para simular um contador, de forma a fazer todas as permutações possíveis entre + e *.

Exemplo para como funcionariam as primeiras 4 interações do loop do contador para 4 bits:
inicializa em 0000:
0001 -> volta pro inicio do while.
0001 -> 0002 -> 0000 -> 0010 -> volta pro inicio do while
0011 -> volta pro inicio do while
0012 -> 0010 -> 0020 -> 0000 -> 0100 -> volta pro inicio do while

A partir deste contador, para cada vez que o valor do vetor op mudasse no while, o programa lê os números do vetor s, para cada número, ele coloca na pilha. Para cada interrogação, ele tira os dois números do topo da pilha, opera neles de acordo com o vetor op e coloca o resultado de volta na pilha. Dessa forma, o gasto de memória é pequeno, contudo, o gasto em processamento é grande.

Analise de Complexidade:

Seja n o numero de números na entrada.

para main.c: a função principal do programa faz a leitura da entrada e chama a próxima função, reverse_polish. A complexidade da leitura, tanto em questão de memória quanto em questão de processamento é a mesma, O(n).

Para reverse_polish.c:
A função reverse_polish conta o número de operadores há na expressão, aloca um vetor de 0s do tamanho adequado e chama a função solve, portanto é O(n) tanto para espaço quanto para processamento.

A função solve faz todas as permutações possíveis entre 0s e 1s para n-1 (já que haverá sempre um operador a menos do que operandos). O calculo de todas as permutações é O(2^(n-1)), como para n grande 1 é irrelevante, podemos dizer que é O(2ˆn). Como o número de permutações é dominante na função solve, pode-se considerar que ela é O(2ˆn) para processamento. Já para

espaço, como a pilha nunca fica maior do que o tamanho de números da expressão, ela é O(n) em questão de memória.

Para stack.c:
createEmptyStack = O(1)
testEmptyStack = O(1)
pushStack = O(1)
popStack = O(1)
destroyStack = O(numero de células na pilha)

Programa como um todo = main + reverse_polish + solve = O(n) + O(n) + O(2^n * n) = **O(2^n * n).**
Gasto de memória = vetor para a expressão, 200 posições + vetor ignorando os espaços + vetor de operações + pilha = O(200) + O(n + (n-1)) (n-1 = numero operadores) + O(n-1) + O(n) -> **O(n)**

Experimentos:

Para fazer testes simples, fiz todos os testes toy entregues para os alunos. Todos rodaram tranquilamente.

Para testes maiores foram usados somente 1s como operandos e valores maiores de entrada. Para esses testes foi usado um incremento de uma interrogação e um 1 a cada vez que o programa rodava.

O maior teste realizado foi um teste com 29 1s e 28 ?s. Todos os testes realizados tiveram os resultados corretos.

Análise de Resultados:

- para uma expressão com 69 caracteres: 0m0.462s
- para uma expressão com 73 caracteres: 0m1.015s
- para uma expressão com 81 caracteres: 0m4.149s
- para uma expressão com 89 caracteres: 0m18.216s

Esses resultados são condizentes com a complexidade do programa.

Conclusão:

O problema apresentado é um problema que muito provavelmente é NP-Completo, uma vez que o calculo de todas as permutações é extremamente complexo. Dessa forma, o tempo demorado para cálculos em valores relativamente grandes é justificado. É necessário encontrar formas de otimizar a solução do problema, mas somente para resolver casos um pouco maiores ou apenas aproximações do resultado correto.