Trabalho Prático 1: Where are the Panzers?

1. Introdução

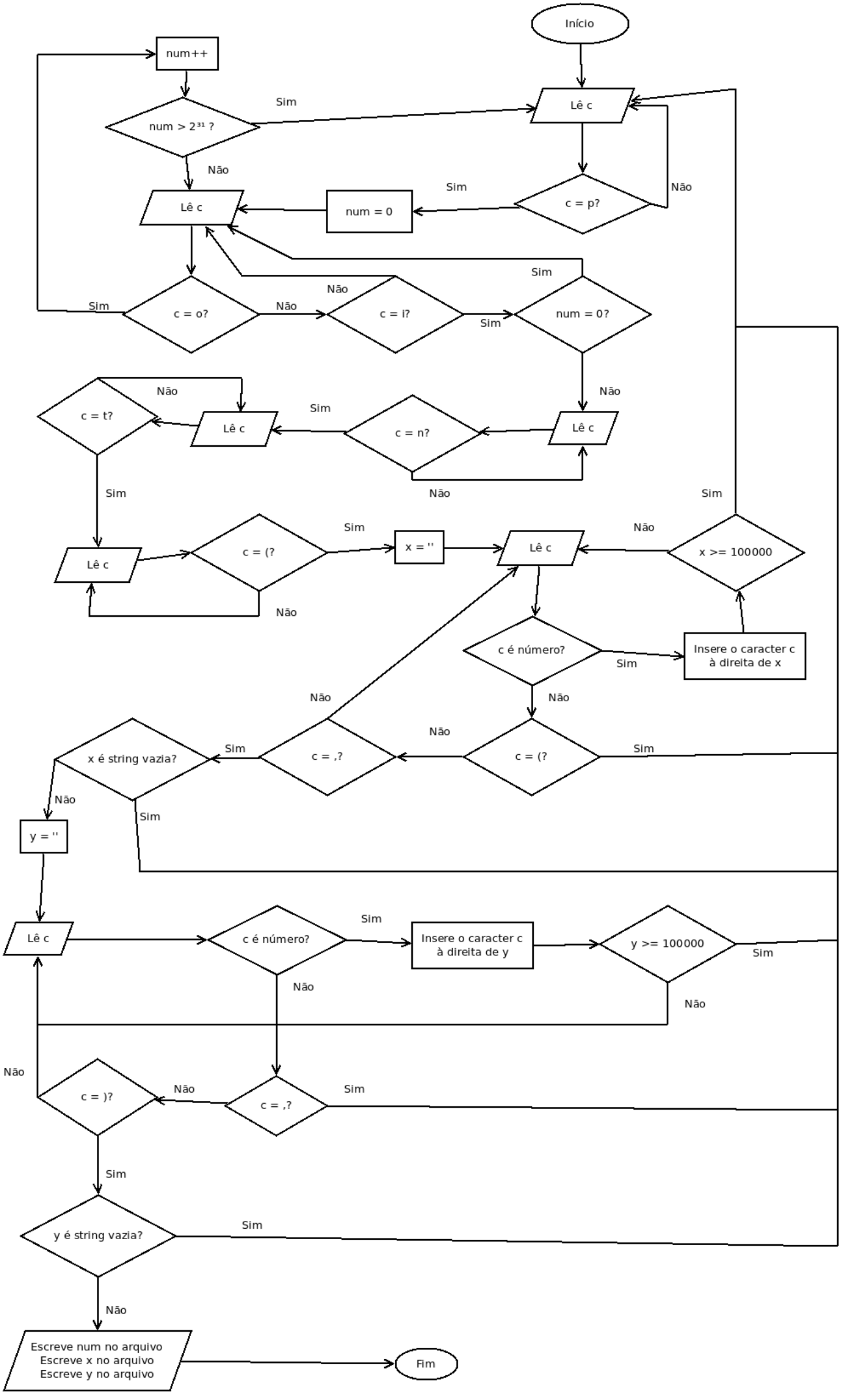
Nesse trabalho temos como objetivo a decodificação de coordenadas de ataques passadas na entrada padrão. O programa deve receber um texto que contenha uma lista de ataques, sendo que cada ataque contém a quantidade de tanques enviados e as suas coordenadas, seguindo um certo padrão. Em todo esse texto, haverão também ruídos, isto é, carateres aleatórios em vários locais, que deverão ser ignorados durante a leitura.

Ao final do processo de decodificação dos ataques, eles deverão ser ordenados para serem exibidos, em ordem de prioridade, da maior para a menor. Um ataque tem maior prioridade que outro quando tem mais tanques, ou em caso de mesmo número de tanques, quando está mais próximo da base, e se a distância da base também for a mesma, o que chegou primeiro tem prioridade superior.

Além disso, o programa deverá rodar com uma quantidade limitada de memória, que será informada na entrada. A quantidade de pontos recebidos pode ser grande, portanto, esses pontos não podem ser simplesmente armazenados em memória primária. Para correto funcionamento do programa, devemos inserir os pontos na memória secundária e ordená-los externamente.

2. Solução do problema

Para efetuar a leitura dos pontos seguindo o padrão especificado, o programa segue o algoritmo dado pelo fluxograma da página seguinte.


Início
num++
Lê c
num > 2³¹ ?
Sim
Não
Lê c
num = 0
Sim
c = p?
Não
Sim
c = o?
Não
Não
c = i?
Sim
num = 0?
Não
c = t?
Não
Lê c
Sim
c = n?
Lê c
Não
Sim
Lê c
c = (?
Sim
x = ''
Lê c
Não
x >= 100000
Sim
Não
c é número?
Sim
Insere o caracter c à direita de x
Não
Não
x é string vazia?
Sim
c = ,?
Não
c = (?
Sim
Não
Sim
y = ''
Lê c
c é número?
Sim
Insere o caracter c à direita de y
y >= 100000
Sim
Não
Não
Não
c =)?
Não
c = ,?
Sim
Sim
y é string vazia?
Sim
Não
Escreve num no arquivo
Escreve x no arquivo
Escreve y no arquivo
Fim

Após a leitura dos pontos, o programa utiliza o arquivo gerado para ordená-los. A ordenação funciona da seguinte maneira: criamos uma lista de pontos na memória principal, usando 80% da memória disponível. Enquanto a quantidade de pontos que ainda faltam ser impressos na saída não couber na memória principal, fazemos o seguinte loop:

- Lemos do arquivo a quantidade de pontos que cabem na memória

- Encontramos o maior ponto dessa lista

- Armazenamos como o maior ponto

- Continuamos lendo o arquivo em blocos que cabem na memória, e trocando o maior ponto sempre que encontramos um ponto maior que ele

- Ao final desse loop, temos o maior ponto, o exibimos na saída, e colocamos valores inválidos na sua posição no arquivo

- Repetimos o processo até que a quantidade de pontos restantes caiba na memória

Agora a quantidade restante de pontos cabe na memória principal. Lemos os pontos restantes do arquivo, e vamos exibindo o maior número da lista na tela e retirando-o até que a lista esteja vazia.

3. Análise de complexidade

Analisaremos a complexidade do algoritmo nos baseando nas variáveis n e m, sendo que n representa a quantidade de pontos lidos e m representa a quantidade de pontos que cabem na memória principal.

Analisando a complexidade em relação à quantidade de vezes que o programa lê todo o arquivo (número de "passadas" no arquivo), podemos concluir o seguinte: o programa lerá o arquivo uma vez para cada ponto, já que ele precisa ler o arquivo inteiro para determinar o maior ponto e em seguida descartá-lo. Isso independe do tamanho da memória, e teremos essa complexidade igual a **O(n)**. O número de acessos ao arquivo será **O(n²)**, pois para cada ponto leremos os n registros uma vez. O número de acessos ao disco será **O(n/m)**.

A complexidade de espaço é **O(n)**, pois armazenamos os pontos em um arquivo apenas.

4. Análise experimental

Para a análise experimental do código, utilizamos arquivos de testes com quantidades variadas de pontos, medindo o tempo levado para a execução do programa inteiro, incluíndo o tempo para leitura dos pontos e para ordenação, de cada arquivo. Para medir o tempo passado, utilizamos a biblioteca <time.h> para contar o número de clocks necessários para executar cada operação, e usamos a variável CLOCKS_PER_SEC para calcular o tempo tomado por cada operação. Os testes foram feitos em uma máquina com processador Core I5 com frequência de 2.6GHz e 4GB de memória RAM.

Foram medidos os valores para as seguintes quantidades de pontos: 500, 1000, 2000, 4000, 8000, 16000, 32000.

O eixo horizontal do gráfico representa a quantidade de milhares de pontos que são tratados, e o eixo vertical representa o número de milissegundos tomados para a execução do programa.

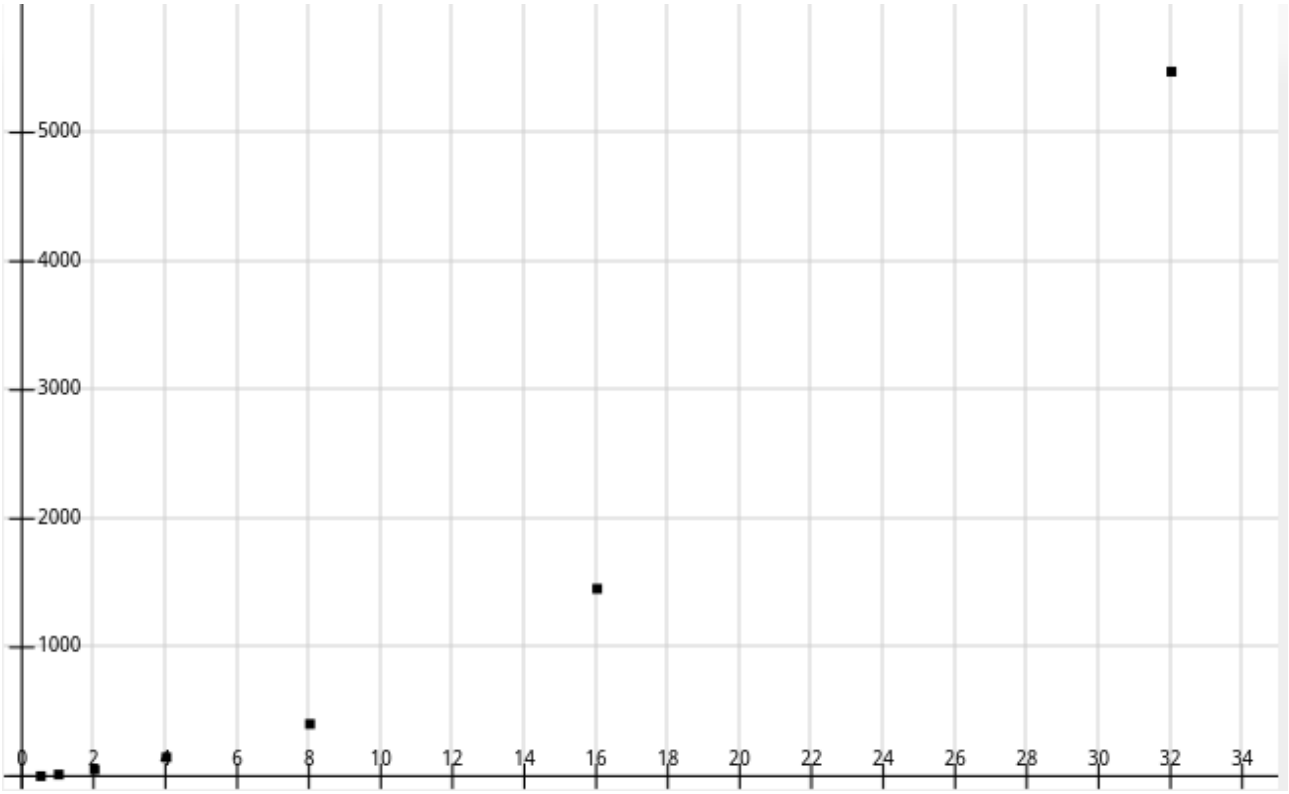


Gráfico de Milhares de pontos x Tempo para a geração dos resultados

Na análise experimental, pudemos perceber que os resultados corroboram com os dados apontados na análise de complexidade, pois os pontos notavelmente se aproximam de uma curva exponencial (n^2, no caso, conforme analisado). Essa aproximação da exponencial faz sentido, visto que a complexidade do programa é basicamente dada pelo número de acessos à memória secundária, pois essa é a operação mais custosa do programa.

5. Conclusão

Após a realização do trabalho, aprendemos a lidar com quantidades massivas de dados, que não cabem na memória principal. Para esse tipo de problema, devemos armazenar os dados na memória secundária, e tratá-los em blocos que cabem na memória. É importante, para a execução desses processos, que hajam poucos acessos ao disco, visto que o acesso ao disco é uma operação muito custosa, sendo ordens de grandeza mais lenta que o acesso à memória principal. A leitura dos dados pôde ser feita de forma simples, semelhante a um autômato.

Na análise experimental, pudemos concluir que a avaliação feita ao longo do trabalho está correta, sendo que o tempo de execução de programa aumenta exponencialmente com um aumento da quantidade de pontos a tratar, pois o número de acessos à memória é dado por n^2, sendo n a quantidade de pontos, e essa é a operação mais lenta do programa.