SCC0220 - Laboratório de Introdução à Ciência da Computação II (2022)

Prof. Diego Furtado Silva

Departamento de Ciências da Computação (SCC)

Instituto de Ciências Matemáticas e de Computação (ICMC)

Universidade de São Paulo

Trabalho 3 - Hora da Ação

Vicenzo D'Arezzo Zilio - 13671790

Felipi Yuri Santos - 11917272

Outubro 2022

1 Introdução ao problema

Tendo em vista que o vencedor do jogo é aquele que faz a última troca, para determiná-lo, será necessário o conhecimento de quem é o primeiro a jogar e quantas são as possibilidades de troca. Interpretado o espaço amostral como uma reta, uma troca é possível quando uma peça maior se encontra à direita de uma peça menor. Logo, a ferramenta para a solução desse problema é a utilização de um algoritmo de ordenação que verifique as adjacências de uma peça, invertendo-as caso necessário.

A resposta mais direta a esse problema é o bubblesort, que percorre o vetor sucessivamente comparando os pares de peças um a um. Porém existe ainda outra alternativa, que é a utilização de um Insertion sort ou um Mergesort.

2 Primeira solução: Bubblesort

2.1 Implementação:

```
int *bubbleSort(int* vetor, int tamanho){
    int i, j;
    int trocou = 0; //Variável que diminui o tempo de execução real do bubbleSort

    for(i = 0; i < tamanho-1; i++){
        for(j = 0; j < tamanho-1-i; j++){ //tamanho-1-i faz reduz o número de comparções do bubble
            if (vetor[j] > vetor[j+1]){
                int aux = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = aux;
                trocou = 1;
            }
        }
        if (trocou == 0) //se não houve qualquer troca, o vetor já está ordenado.
            break;
    }
    return vetor;
}
```

2.2 Complexidade

O melhor caso para o cálculo da complexidade do BubbleSort ocorre quando o vetor já está ordenado. Caso isso aconteça, o código para o algoritmo irá percorrer o vetor inteiro uma única vez (graças a utilização da variável do tipo flag "trocou"), ou seja, irá efetuar n operações de comparação e nenhuma operação de troca.

Para a análise do pior caso da complexidade do BubbleSort, temos de tomar o vetor ordenado de forma inversa ao ordenamento pedido. Por exemplo, se queremos uma ordenação crescente, o pior caso ocorrerá quando o vetor de input estiver ordenado de forma decrescente. Sendo assim, no primeiro passo da execução do algoritmo, teremos $n-1$ operações de comparação (e de trocas, considerando o cenário do vetor inversamente ordenado), no passo 2, teremos $n-2$ operações de comparação, no passo 3, $n-3$, e assim por diante até chegar no passo $n-1$, onde executaremos apenas uma operação de comparação, a última do algoritmo.

Somando todas as contagens de operações teremos $(n-1)+(n-2)+(n-3)+...+(n-n+1)$. Esse tipo de soma pode ser resolvido utilizando a fórmula de soma de uma Progressão Aritmética $S_n = \frac{n}{2}(a_n + a_1)$. Essa soma resultará nessa contagem de operações:

$$f(n) = \frac{n(n+1)}{2} \tag{1}$$

É evidente que para essa contagem de operações teremos Big-O: $O(n^2)$ para o BubbleSort.

3 Segunda solução: Mergesort

3.1 Implementação:

A implementação da versão mais complexa se encontra do arquivo enviado através do *Run.Codes*.

3.2 Complexidade:

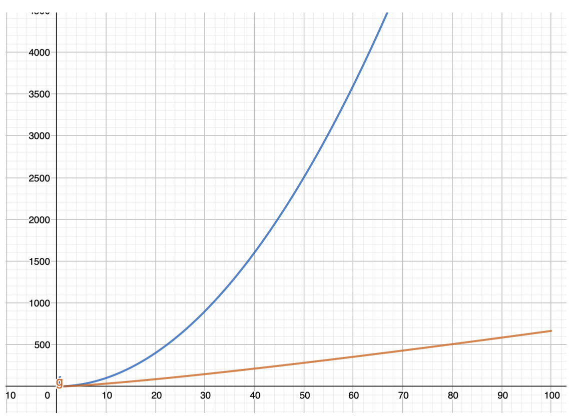
O mergesort é um algoritmo que não possui diferenciação entre melhor e pior caso, logo, sua complexidade varia em função do tamanho de entrada (n):

$$\forall n \in N \Longrightarrow f(n) = O(n \cdot \log_2 n); \tag{2}$$

4 Análise comparativa:

$$Laranja = O(n \cdot \log_2 n) - Mergesort; \tag{3}$$

$$Azul = O(x\hat{}2) - Bubblesort; \tag{4}$$



5 Conclusão e comentários finais:

Partindo da noção de que este é um problema de ordenação, o impasse do projeto foi adaptar os algoritmos já conhecidos à situação proposta pelo jogo. A premedição das regras de inversão são construídas em cima de um par ajacente de gomas, logo, necessita-se de um método em que a comparação ocorra entre elementos adjacentes.

Para o Bubblesort, solução mais didática, tal natureza já é contemplada pelo método, que, no segundo, passo realiza uma condicional referente a pares adjascentes, assim como no jogo. Entretanto, para o MergeSort, a comparação é adjascente somente nos casos base da recursão - vetor de 2 unidades. Por fim, para adaptar sua implementação, utilizamos uma aritimética durante o incremento do contador: ao ser contabilizada uma troca, a variável recebe não somente um incremento, mas também o módulo de uma distância, que, por sua vez, significa a distância que tal elemento tera que, virtualmente, percorrer até chegar em sua devida posição.