Discente: Ronaldo Ribeiro Porto Filho – 202410131
Data: 21/09/2025

Exercício complexidade de tempo do algoritmo Quicksort

Vamos analisar a complexidade de tempo do algoritmo Quicksort para o pior caso, o qual representa a notação Big-O do mesmo. Ele ocorre quando o vetor está ordenado de forma decrescente, e por conta disso, gera partições desequilibradas, fazendo diversas comparações e trocas a cada chamada.

```c
void quicksort(int *vet, int esq, int dir){
    if (esq >= dir){
        return;
    }
    int indexPivo = particiona(vet, esq, dir);
    quicksort(vet, esq, indexPivo-1);
    quicksort(vet, indexPivo+1, dir);
}

int particiona (int *vet, int esq, int dir){
    int pivo = vet[esq];
    int left = esq + 1;
    int right = dir;
    int aux;

    while (1){
        while (left <= right && vet[left] <= pivo){
            left++;
        }
        while (right >= left && vet[right] > pivo){
            right--;
        }
        if (left > right){
            break;
        }
        aux = vet[left];
        vet[left] = vet[right];
        vet[right] = aux;
    }

    vet[esq] = vet[right];
    vet[right] = pivo;
    return right;
}
```

Primeiro, vamos analisar o custo da função particiona: Dentro do laço while mais externo temos 2 outros laços while, um if e uma sequência de troca. Assim, basta analisar o tempo dentro desse laço mais externo, pois as outras atribuições antes e após ele são constantes em toda chamada, não mudando nunca.

No primeiro laço while interno, no pior caso, ele percorre todo o vetor de tamanho n, parando quando a verificação left <= right não for mais verdadeira, e no segundo laço interno, ele para logo de início, pois a verificação right >= left já não é verdadeira. Ou seja, o algoritmo entra no if (left > right) e para a execução do laço while mais externo, e por fim, realiza a troca final colocando o pivô que estava na primeira posição do vetor em seu índice correto, a última posição.

Com isso, podemos ver que o algoritmo do particiona possui complexidade de tempo linear, ou seja, $O(n)$, pois ele sempre percorre todo o vetor de tamanho n toda vez que é chamado. No pior caso, ele percorreu todo o vetor da esquerda pra direita, e em casos intermediários (a maioria dos casos) ele percorre todo o vetor também, porém, aproximadamente metade no primeiro while e a outra metade no segundo while, realizando trocas ao longo do caminho, parando quando as duas variáveis de controle se encontram mais ou menos no meio do vetor, e assim, saindo do laço mais externo.

Dessa forma, vamos analisar agora o algoritmo quicksort no pior caso de fato. Inicialmente ele possui um if, depois chama a função particiona e depois ocorre duas recorrências, que podem ser expressas dessa forma:

$$T(n) = T(0) + T(n - 1) + T(p) \text{ , } n > 1$$

Onde $T(p)$ é a complexidade da função particiona, ou seja, $O(n)$, e como após o primeiro particiona o índice correto do pivô ficou na última casa do vetor, somente uma recorrência terá valor de verdade. Pois o subvetor direito que seria analisado tem tamanho 0, tendo assim, um valor constante negligenciável, e o subvetor esquerdo tem tamanho $n - 1$, esse sim tendo um valor significante.

Desenvolvendo essa recorrência:

Caso Base:

$$T(1) = 1$$

Recorrência:

$$T(n) = T(0) + T(n - 1) + T(p) \text{ , } n > 1$$

$$T(n) = T(n - 1) + n \text{ , } n > 1$$

$$T(n) = (T(n - 2) + n) + n \text{ , } n > 1$$

$$T(n) = ((T(n - 3) + n) + n) + n \text{ , } n > 1$$

$T(n) = (((T(n - 4) + n) + n) + n) + n \ , \ n > 1$

$T(n) = (((T(n - 4) + n) + n) + n) + n \ , \ n > 1$

$T(n) = T(n - 4) + 4n \ , \ n > 1$

$T(n) = T(n - k) + kn \ , \ n > 1$

Essa recorrência termina quando a função T(n) ser o caso base T(1), ou seja,

$n - k = 1$

$k = n - 1$

Substituindo:

$T(n) = T(n – n + 1) + (n - 1)*n \ , \ n > 1$

$T(n) = T(1) + n^2 - n \ , \ n > 1$

$T(n) = n^2 - n + 1 \ , \ n > 1$

Termo de maior ordem: n^2

$T(n) \in O(n^2)$

Assim, encontramos a complexidade de tempo (classe assintótica) do algoritmo Quicksort para o pior caso, $O(n^2)$.

Por fim, como cada chamada recursiva do algoritmo Quicksort no pior caso reduz o tamanho do vetor em 1 (T(n-1)), como provado anteriormente, então sua pilha de execução é de tamanho n, pois a cada recursão é gerada apenas uma nova pilha de execução, sendo a outra de tamanho vazia, e ordenando somente um índice do vetor, diminuindo a cada chamada apenas 1 elemento.

Prova:

Caso base:

$T(1) = 1$

Recorrência:

$T(n) = T(n - 1) + 1 \ , \ n > 1$

$T(n) = (T(n - 2) + 1) + 1 \ , \ n > 1$

$T(n) = ((T(n - 3) + 1) + 1) + 1 \ , \ n > 1$

$T(n) = (((T(n - 4) + 1) + 1) + 1) + 1 \ , \ n > 1$

$T(n) = T(n - 4) + 4 \ , \ n > 1$

$T(n) = T(n - k) + k \ , \ n > 1$

$n - k = 1$

$k = n - 1$

$T(n) = T(n - n + 1) + n - 1 \text{ , } n > 1$

$T(n) = T(1) + n - 1 \text{ , } n > 1$

$T(n) = n \text{ , } n > 1$

Termo de maior ordem: n

$T(n) \in O(n)$