Portfólio – Algoritmo e Complexidade

Gabriel Gian, 3° período

Quick sort

O quick sort é um algoritmo onde elementos de uma coleção são ordenados de maneira crescente ou decrescente, lendo-se a sequência por meio de dois vetores, `high` e `low`, e um pivô. O pivô é tomado como ponto de partida e os vetores `high` e `low` são respectivamente o início e o fim da coleção. Eles são trocados de posição a partir das iterações do algoritmo que, por meio de um predicado, checam o valor do elemento em questão e comparam com o do vetor – executando a troca caso o valor seja maior, no caso do vetor `high`, ou menor, no caso do vetor `low`.

Código em Python

```
def particao(array, start, end):

    pivo = array[start]
    low = start + 1
    high = end

    while True:

        while low <= high and array[high] >= pivo:
            high -= 1

        while low <= high and array[low] <= pivo:
            low += 1

        if low <= high:
            array[low], array[high] = array[high], array[low]

        else:
            break

    array[start], array[high] = array[high], array[start]

    return high
```

Primeiramente, nesse algoritmo de quick sort para a ordenação crescente, é definida a função `particao`. Essa função secundária será usada pela principal, `quicksort`. A função `particao` opera tomando-se como parâmetros um array, um valor de início e um valor de fim. Ela define o valor de início como o pivô, estabelece o elemento seguinte como o valor de `low`, e o valor do parâmetro final como o valor de `high`.

Em seguinte, por meio de um laço `while True`, é checado, por meio de um laço `while`, se o valor de `low` é menor ou igual o valor de `high` e o valor de `high` como elemento do array é maior que o pivô (isso impede que excedamos o valor do `low`, o que significaria que já foram movidos todos os valores menores para a esquerda, ordenando-os em ordem crescente). Caso seja, o valor de `high` se torna referente ao elemento anterior a ele. O mesmo é feito, embora de maneira inversa, no laço `while` seguinte, responsável por checar os valores dos elementos em relação ao vetor `low`.

Por último, temos um predicado que verifica se o valor de `low` é menor ou igual ao de `high`, o que significaria que o elemento deve ser trocado. É então executada um operação de troca entre os valores de `low` e `high` no array. Caso contrário, o laço é quebrado. Por fim, é trocado o pivô com o valor de `high` e retornado o valor de `high`.

```
def quicksort(array, start, end):

    if start >= end:
        return None

    p = particao(array, start, end)
    quicksort(array, start, p-1)
    quicksort(array, p+1, end)
```

No algoritmo quicksort, é tomado como parâmetros os mesmos valores requeridos por `particao`, já que serão passados a essa mesma função. Caso o valor de `start` seja maior ou igual ao de `end`, a função retorna `None` e não executa nenhuma operação. Caso contrário, tomamos uma variável p como o valor de retorno da função `particao`, que é o seu valor `high`, e aplicamos a função `quicksort` recursivamente na mesma, embora de duas maneiras: a primeira tomando como valor de `end` o valor de p decrescido de 1. Isso faz com que o algoritmo não releia o valor de `high` anterior já que ele se encontra ordenado. O oposto acontece na segunda ordenação, onde tomamos como `start` o valor de p acrescido de 1.

```
array = [6, 50, 3, 7, 72]

quicksort(array, 0, len(array) - 1)
print(array)

>> [3, 6, 7, 50, 72]
```

Complexidade

A complexidade em notação Big O do quick sort é $O(n^2)$. Isso ocorre no pior caso em que um elemento já se encontra ordenado mas acaba gastando computação para a checagem. Todavia, a complexidade, *em média*, acaba sendo O(n log (n)), pois geralmente, o pivô acaba sendo movido para a metade do array, e os pivôs de ambos os lados tendem a serem movidos para os meios de ambas as metades, o que assegura a ordenação e aumenta a eficiência em termos de complexidade.

Bogosort

O bogosort, ou permutation sort, é um algoritmo extremamente ineficiente, e de alta complexidade, que, essencialmente, gera uma permutação aleatória ou determinística de uma sequência de elementos em um laço, que se repete até que a sequência esteja ordenada. Sua implementação é relativamente fácil.

Código em Python

```
import random

def ordenadop(array):
     n = len(array)
     for i in range(0, n-1):
          if (array[i] > array[i + 1] ):
               return False
     return True

def permutar(array):
     n = len(array)
     for i in range(0, n):
          r = random.randint(0, n-1)
          array[i], array[r] = array[r], array[i]

def bogosort(array):
     while (ordenadop(array) == False):
          permutar(array)
```

Nesse exemplo, usaremos o modo mais fácil de se implementar esse tipo de algoritmo – o modo randômico. O modo determinístico necessitaria a implementação de um aparato mais complexo para a geração de permutações, dificultando sua implementação. Começamos por importar a biblioteca `random`. Depois, implementamos um método `ordenadop`, responsável por checar se o array já se encontra ordenado. Este predicado faz uma checagem simples entre dois elementos do array para checar

se estão na ordem correta ao se comparar cada um com o seu sucessor na sequência e retorna `True` ou `False` de acordo com o seu estado.

Para gerar as permutações, definimos a função permutar, que recebe um array como parâmetro e e define uma variável n como o tamanho desse array. Em um laço `for`, de um intervalo do primeiro elemento a um elemento a mais que o total de elementos (isso evita permutações erradas), criamos uma variável com o valor gerado por um método de número inteiro que pode ser gerado num intervalo de 0 ao último elemento do array. Após o número aleatório ser gerado, o valor armazenado na posição correspondente àquela iteração é trocado pelo valor armazenado na posição do número aleatório gerado.

Finalmente, no método `bogosort`, tomando um array, checamo-no com o predicado `ordenadop`, e enquanto, por meio de um laço `while`, esse retorne `False` – para não ordenado, o método permutar é aplicado, engendrando uma nova permutação até que o array esteja ordenado.

Complexidade

A complexidade do bogosort, no melhor caso, a complexidade é O(n), na ínfima (e improvável) chance do algoritmo levar apenas uma tentativa para ordenar o array corretamente e executar a checagem. Já o pior caso depende da implementação do algoritmo. Em algoritmos bogosort implementados de maneira determinística, a complexidade chega a atingir o impraticável O((n+1)!). Todavia, essa esquálida complexidade chega a ser privilegiada perto da complexidade apresentada na implementação randômica do bogosort, embora seja semelhante no que concerne ao tempo de execução, que impede suas conclusões catastróficas, teoricamente possui uma complexidade indefinível, de modo que não pode ser restringida.