Estruturas de Dados - Quicksort

Aula passada

O que vimos:

- Merge Sort

Complexidades:

Algoritmo	melhor caso	pior caso
Merge Sort	$O(n \log n)$	$O(n \log n)$

Quick Sort

- Ordenação por comparação
- Assim como o Merge Sort, também faz ordenação por partes
- Também é um algoritmo de divisão e conquista
- Faz comparações com um certo elemento do vetor

Quick Sort

- Ideia para ordenar:
  - caso tenhamos um vetor unitário, trivialmente ele já está ordenado
  - selecionar, aleatoriamente ou não, um elemento x do vetor
    - chamaremos esse elemento de pivô
  - particionar: reposicionar os elementos do vetor de modo que:
    - os elementos menores que x tenham índice no vetor menor que o pivô
    - os elementos maiores que x tenham índice no vetor maior que o pivô

Quick Sort

- Ideia para ordenar (continuação):
  - resultado:
    - pivô x em uma certa posição no vetor
    - todos os elementos menores que x em S estarão a esquerda do pivô
    - todos os elementos maiores que x em S estarão a direita do pivô
  - a partir disso, basta ordenarmos os dois lados
    - ordenamos o subvetor de menores e o subvetor de maiores
- Os passos listados são repetidos para a ordenação de cada subvetor
- Corretude: essa ideia funciona?

Quick Sort

- Caso base da recursão:
  - vetor unitário: caso trivial onde o vetor é ordenado
- Chamadas recursivas:
  - divisão e conquista: consideramos que as chamadas recursivas ordenam os subproblemas com sucesso
- Particionamento do vetor:
  - separar os elementos do vetor em maiores ou menores que o pivô
  - este caso não pode ser resolvido com chamadas recursivas; se trata de um novo problema
  - caso seja resolvido corretamente, torna o algoritmo correto

Quick Sort: função Partition

- Função Partition:
  - recebe um vetor arbitrário S, seleciona um pivô x e reposiciona os elementos de S de modo que
    - elementos menores que x estarão a esquerda do pivô
    - elementos maiores que x estarão a direita do pivô
  - como fazer?

Quick Sort: função Partition

- Função Partition:
  - recebe um vetor arbitrário S, seleciona um pivô x e reposiciona os elementos de S de modo que
    - elementos menores que x estarão a esquerda do pivô
    - elementos maiores que x estarão a direita do pivô
  - ideia:
    - procurar um elemento a a partir da esquerda do vetor S, que seja maior que x
    - procurar um elemento b a partir da direita do vetor S, que seja menor que x
    - trocar a posição de a e b em S

Quick Sort: função Partition

Algoritmo: Partition(S, p, r)

Entrada: vetor S, índices p e r
Saída: posição q do pivô em S

```
1  piv = S[p]
2  i = p
3  j = r
4  enquanto i ≤ j faça
5  |   enquanto S[i] ≤ piv faça
6  |   |   i = i + 1
7  |   enquanto S[j] > piv faça
8  |   |   j = j − 1
9  |   se i < j então
10 |   |   trocar S[i] com S[j]
11 S[p] = S[j]
12 S[j] = piv
13 retorne j
```

Quick Sort: algoritmo

Algoritmo: QuickSort(S, p, r)

Entrada: vetor S, índices p e r

Saída: vetor S ordenado de p a r

```
1 se p < r então
2 |   q = Partition(S, p, r)
3 |   QuickSort(S, p, q − 1)
4 |   QuickSort(S, q + 1, r)
```

Quick Sort: complexidade

- Complexidade do Quick Sort:
  - cada chamada para QuickSort passando um vetor de tamanho n inclui
    - uma chamada para Partition, que tem complexidade $O(n)$
    - duas chamadas para QuickSort passando vetores de tamanho $\frac{n}{2}$
  - análise de complexidade: semelhante ao MergeSort
  - complexidade: $O(n \log n)$
- Complexidade do QuickSort no melhor caso e caso médio:
  - duas chamadas para QuickSort passando vetores de tamanho médio igual a $\frac{n}{2}$
  - complexidade: $O(n \log n)$

Quick Sort: complexidade

- Complexidade do Quick Sort no pior caso:
  - pior caso: pivô selecionado em cada nível de recursão é sempre o maior (ou menor) elemento do vetor
    - uma chamada para Partition, que tem complexidade $O(n)$
    - duas chamadas para QuickSort passando vetores de tamanho 0 (desprezada) e $n-1$
  - análise de complexidade: análoga ao BubbleSort ou InsertionSort
  - complexidade: $O(n^2)$

Quick Sort

- Vantagens:
  - implementação relativamente simples (Partition é mais fácil de implementar do que Merge)
  - complexidade $O(n \log n)$ no caso médio
  - pouco uso de memória auxiliar
    - utiliza uma quantidade constante de variáveis auxiliares, seja qual for o tamanho n do vetor
  - método rápido (o laço interno é simples)
- Desvantagens:
  - não é estável
  - problema: seleção do pivô
  - pior caso: $O(n^2)$

Quick Sort: escolha do pivô

- Vimos que a escolha do pivô influencia na complexidade
  - um bom pivô resulta numa complexidade $O(n \log n)$
  - um mau pivô resulta numa complexidade $O(n^2)$
- Problema: qual elemento escolher como pivô?

Quick Sort: mediana de um vetor

- Problema: qual elemento escolher como pivô?
- Resposta direta: um elemento que "corte o vetor no meio"
- Resposta: mediana!
  - mediana de um vetor S: um elemento x tal que metade dos elementos sejam maiores ou iguais a x, e metade dos elementos sejam menores ou iguais a x
  - "divide o vetor no meio"
- A princípio, considere um vetor ordenado S de tamanho n
  - se n é ímpar, x é o valor na posição $\frac{n+1}{2}$ de S
  - se n é par, x é a média dos valores nas posições $\frac{n}{2}$ e $\frac{n}{2}+1$

Quick Sort: mediana de um vetor

- Para vetores ordenados, encontrar a mediana é $O(1)$
  - se o tamanho do vetor for ímpar, basta checar a posição do meio
  - se o tamanho do vetor for par, fazemos a média entre os valores nas duas posições do meio
- Problema: como calcular a mediana para um vetor S arbitrário?
- Solução simples: ordenar o vetor e calcular a mediana
  - ordenar: $f(n)$ (complexidade do método de ordenação)
  - calcular a mediana: $O(1)$
  - complexidade: $f(n)$
- Conseguimos fazer melhor?
  - sim, mas não trabalharemos isso a fundo
  - porém, é importante saber um pouco sobre mediana