Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

29 de junho de 2016

Algoritmos de Classificação

Algoritmos de Classificação Gerais

- Estes algoritmos se aplicam a uma sequência de valores, em estrutura linear indexada.
- O índice é importante na maioria dos algoritmos.
- Alguns algoritmos são baseados em trocas: Ordenação por Bolha (Bubble Sort), Ordenação Rápida (Quick Sort).
- Alguns são baseados em seleção: Ordenação por Seleção (Select Sort), Ordenação por Combinação (Merge Sort).
- Alguns são baseados em inserção: Ordenação por Inserção (Insert Sort), Ordenação por Inserção de Passo (Shell Sort), Ordenação por Árvore Binária de Busca, Ordenação via estrutura de Heap (Heap Sort).
- Existem outros algoritmos que usam o fato de alguma restrição nos números, por exemplo, quantidade de casas decimais fixa.

Bubble Sort: Técnica

- Varre-se a estrutura comparando elementos 2 a 2. Se estiver fora de ordem, troca-os.
- Na primeira vez, o maior elemento acaba em sua posição final.
- Na segunda vez, o segundo maior, e assim por diante.
- Então é necessá8rio varrer *n* vezes a estrutura, 1 par na primeira vez, 2 pares na segunda, ... n-1 pares na última.
- Eventualmente pode-se parar se em alguma varredura não haver sido realizada nenhuma troca.

Bubble Sort: O algoritmo

Entrada: : Uma sequência A e seu tamanho n.
Saída: : A sequência A classificada em ordem não decrescente.

Algoritmo BUBBLE_SORT(A,n)
  para $i \leftarrow 1$ **até** n **faça**
    para $j \leftarrow 1$ **até** $n - i$ **faça**
      se $A[j] < A[j+1]$ **então**
        $A[j] \leftrightarrow A[j+1]$

Bubble Sort: Calculando o Tempo

- São duas somas (loops for), dentro dos loops, no máximo 1 troca.

$$\sum_{i=1}^{n}\sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n}(n-i) = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

- Podemos dizer que o algoritmo é $O(n^2)$

Quick Sort: A técnica

- Considerando um vetor inicial $A[p..r]$, vamos dividi-lo em duas partes: $A[p..q-1]$ e $A[q+1..r]$, tais que:

	p	q	r
A	$\leqslant\ x$	x	$>\ x$

$$A[p..q-1] \leqslant A[q] < A[q+1, p]$$

- O elemento $A[q]$ já está em sua posição final. Vamos aplicar recursivamente o algoritmo nos subvetores $A[p..q-1]$ e $A[q+1..r]$. Na recursão, se $p \geqslant q-1$ ou $q+1 \geqslant r$ não há nada a fazer.
- No final o vetor estará ordenado.

Quick Sort: Partição

Problema:

Rearranjar um dado vetor $A[p..r]$ e devolver um índice q, $p \leqslant q \leqslant r$ tais que:

$$A[p..q-1] \leqslant A[q] < A[q+1..r]$$

Exemplo:

Entrada:

	p									r
A	5	3	7	5	9	4	8	2	7	6

Saída:

	p					q				r
A	5	3	5	4	2	6	8	9	7	7

Particionando

	p									r
A	5	3	7	5	9	4	8	2	7	6
i	j									x
A	5	3	7	5	9	4	8	2	7	6
	i	j								x
A	5	3	7	5	9	4	8	2	7	6
		i	j							x
A	5	3	7	5	9	4	8	2	7	6
		i		j						x
A	5	3	5	7	9	4	8	2	7	6
			i		j					x
A	5	3	5	7	9	4	8	2	7	6
			i			j				x

Particionando

A	5	3	5	7	9	4	8	2	7	6
			i			j				x
A	5	3	5	4	9	7	8	2	7	6
				i			j			x
A	5	3	5	4	9	7	8	2	7	6
				i				j		x
A	5	3	5	4	2	7	8	9	7	6
					i				j	x
A	5	3	5	4	2	7	8	9	7	6
					i					jx
A	5	3	5	4	2	6	8	9	7	7
	p					q				r

Quick Sort: Algoritmo Particione

Algoritmo PARTICIONE(A,p,r)

$x \leftarrow A[r]$ ▷ x é o "pivô"

$i \leftarrow p - 1$

para $j \leftarrow p$ **até** $r - 1$ **faça**

  se $A[j] \leqslant x$ **então**

    $i \leftarrow i + 1$

    $A[i] \leftrightarrow A[j]$

$A[i+1] \leftrightarrow A[r]$ **retorne** $i + 1$

Quick Sort: O Algoritmo

Algoritmo QUICK_SORT(A, p, r)
  se $p < r$ **então**
    $q \leftarrow PARTICIONE(A, p, r)$
    $QUICK\_SORT(A, p, q-1)$
    $QUICK\_SORT(A, q+1, r)$

- Vamos calcular o tempo para o algoritmo.
- É simples de observar que para o PARTICIONE é $O(n)$
- Mas como fica para o QUICK_SORT? Não sabemos onde será a partição.

Complexidade do QuickSort: Pior Caso

- Vamos supor que o conjunto já está ordenado.
- O pivo está sempre em sua posição final (a última).
- A cada partição, ficamos com uma parte com (n-1) elementos, e o pivô, a outra parte é vazia:
- Teremos de rodar *n* vezes o algoritmo particione, cada vez com um elemento a menos, que ó mesmo caso do Bubble, ou seja, no pior caso o Quick Sort é $O(n^2)$.

Complexidade do QuickSort: Melhor caso e caso médio

- Vamos considera que em todas as partições o pivô termina no meio da partição.
- A cada divisão, rodamos o Particione em cada uma das partes (exceto no pivo).
- Vamos dividir sempre em dois: 2 partes, 4 partes, ...
- Faremos $\log n$ divisões, e rodamos o particione em todas partes a cada divisão, logo teremos que no melhor caso, o quick sort é $O(n \log n)$.
- No caso médio a divisão não está no meio, mas sempre temos duas partes. Teremos uma quantidade logaritmica de divisões, da mesma forma. No caso médio, também é $O(n \log n)$.
- Não faremos neste nível uma prova formal do cálculo destes tempos.

Select Sort: A técnica

- Varremos a sequência e selecionamos o menor elemento.
- Trocamos ele de posição com o primeiro elemento.
- Iterativamente continuamos a operação com a sequência começando no próximo elemento.

Select Sort: O algoritmo

Entrada: Sequência A de números e tamanho n
Saída: Sequência com os mesmos números de forma ordenada

Algoritmo $\mathrm{SELECT\_SORT}$(A,n)
  para $i \leftarrow 1$ **até** $n - 1$ **faça**
    $min \leftarrow i$
    para $j \leftarrow i + 1$ **até** n **faça**
      se $A[j] < A[min]$ **então**
        $min \leftarrow j$
    $A[i] \leftrightarrow A[min]$

Select Sort: Calculando o tempo

- São duas somas (loops for) dentro delas no máximo 1 atribuição do *min*

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1)=\sum_{i=1}^{n-1}(n-i)=n(n-1)-\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

- Igualmente ao Bubble, o Select Sort é $O(n^2)$.

Merge Sort: A técnica

- Dividimos a seqüência em duas partes
- Recursivamente aplicamos o algoritmo em cada parte, e cada parte estará ordenada.
- Intercalam-se os valores resultantes de cada parte, formando uma seqüência final ordenada.
- A base da recursão ocorre quando existe um único elemento, ou a sequência é vazia, neste caso, nada a fazer.

Merge Sort: O algoritmo

Entrada: Uma sequência de números, sendo p o índice de início e r de fim
Saída: A mesma sequência de números, ordenada

Algoritmo MERGE_SORT(A,p,r)
 se $p < r$ **então**
  $q \leftarrow (p + r)/2$
  MERGE_SORT(A, p, q)
  MERGE_SORT$(A, q + 1, r)$
  INTERCALA(A, p, q, r)

Merge Sort: Rotina INTERCALA

Entrada: Uma sequência onde no intervalo [p,r] a sequência está ordenada no subarranjo [p,q] e no subarranjo [q,r].

Saída: A mesma sequência, ordenada no intervalo [p,r].

Algoritmo $\mathrm{INTERCALA}$(A,p,q,r)

```
para i ← p até q faça
    B[i] ← A[i]
para j ← q + 1 até r faça
    B[r + q + 1 − j] ← A[j]
i ← p
j ← r
para k ← p até r faça
    se B[i] ⩽ B[j] então
        A[k] ← B[i]
        i ← i + 1
    senão
        A[k] ← B[j]
        j ← j − 1
```

Merge Sort: Calculando o Tempo

- Obviamente a rotina intercala é $O(n)$.
- No Merge Sort, dividimos em dois, em cada divisão aplicamos intercala em todo o intervalo.
- O número de divisões é $\log n$, em cada divisão temos intercala.
- O Merge Sort é $O(n \log n)$

Insert Sort: A técnica

- A partir do segundo elemento, nós vemos, com relação aos elementos de trás na sequência, em que posição ele se enquadra.
- Ao inserir o elemento da posição *i* em seu lugar correto entre os *i* primeiros elementos, estes estarão em ordem.
- Termina quando inserirmos o último elemento na posição correta.

Insert Sort: O algoritmo

Entrada: Uma sequência A e seu tamanho n
Saída: A sequência A ordenada

Algoritmo $\textsc{Insert\_Sort}(A, n)$
  para $i \leftarrow 2$ **até** n **faça**
    $chave \leftarrow A[i]$
        $\triangleright$ Inserir $A[i]$ na seqüência ordenada $A[1 \cdot \cdot i-1]$.
    $j \leftarrow i-1$
    enquanto $j > 0$ e $A[j] > chave$ **faça**
      $A[j+1] \leftarrow A[j]$
      $j \leftarrow j-1$
    $A[j+1] \leftarrow chave$

Insert Sort: Calculando o tempo

- No melhor caso, está ordenado, então o loop em j não é realizado, é $O(n)$
- São duas somas, no pior caso, a segunda soma somente para com $j = 0$.
- Em cada soma uma comparação

$$\sum_{i=2}^{n}\sum_{j=1}^{i-1}(1) = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} - 1 - (n-1) = \frac{n^2}{2} - \frac{3n}{2} - 2$$

- O caso médio é tão ruim quanto o pior caso, este algoritmo é $O(n^2)$

Shell Sort: A técnica

- Shell Sort é uma modificação do Insert Sort.
- São criados passos largos na comparação da posição a ser inserida
- Estes passos vão diminuindo e termina com o passo 1 que é o próprio Insert Sort.
- A idéia é tentar jogar rapidamente elementos pequenos no começo da sequência com o mínimo de iterações no loop.
- Os passos escolhidos é importante que não sejam múltiplos um dos outros, pois acaba não sendo producente.
- Um exemplo de passos são números primos.
- Este algoritmo não altera o tempo do Insert Sort, é $O(n^2)$

Shell Sort: O algoritmo

Entrada: Uma sequência A e seu tamanho n, um conjunto de passos P

Saída: A sequência A ordenada

Algoritmo $\textsc{Insert\_Sort}(A, n, P)$

para $p \leftarrow tamanho(P)$ **até** 1 **faça**
  para $i \leftarrow P[p] + 1$ **até** n **faça**
    $chave \leftarrow A[i]$
    $j \leftarrow i - P[p]$ ▷ O passo em j
    enquanto $j > 0$ e $A[j] > chave$ **faça**
      $A[j + P[p]] \leftarrow A[j]$
      $j \leftarrow j - P[p]$
    $A[j + 1] \leftarrow chave$

Binary Search Tree Sort: A técnica

- Árvore Binária de Busca utiliza uma classificação de ordem ao inserir um elemento.
- Dado um raiz da árvore ou subárvore:
  - Todos os descendentes do lado esquerdo tem valores menor ou igual ao valor da raiz.
  - Todos descendentes do lado direito tem valores maior que o da raiz.
- A inserção de um elemento se dá a partir do raiz da árvore.
  - Se o elemento for menor ou igual, tenta-se inserir no filho esquerdo.
  - Se o elemento for maior, tenta-se inserir no filho direito.
- O elemento é inserido quando atinge uma árvore vazia.
- A ordenação final se dá em uma varredura em ordem simétrica.

Binary Search Tree Sort: O algoritmo - Inserindo

Algoritmo $\textsc{insValor}(raiz, v)$

- $p = \texttt{criarArvore}(v)$
- **se** $raiz = \emptyset$ **então**
  - $raiz = p$
- **senão**
  - **se** $v \leqslant \texttt{info}(raiz)$ **então**
    - $\texttt{insValor}(\texttt{filhoEsq}(raiz), v)$
  - **senão**
    - $\texttt{insValor}(\texttt{filhoDir}(raiz), v)$

Binary Search Tree Sort: O algoritmo - Varrendo

Algoritmo $\textsc{inOrdem}(raiz, A, i)$
  se $raiz \neq \emptyset$ **então**
    inOrdem($raiz- > filhoEsq(), A, i$)
    $A[i] \leftarrow raiz- > val$
    $i \leftarrow i + 1$
    inOrdem($raiz- > filhoDir(), A, i$)

Binary Search Tree Sort: O algoritmo

Entrada: Sequência A e seu tamanho n
Saída: A sequência A com seus elementos ordenados

Algoritmo BST_SORT(A,n)
  BST T
  para $i \leftarrow 1$ **até** n **faça**
    insValor$(T, A[i])$
  $i \leftarrow 1$
  inOrdem$(raiz, A, i)$

Binary Search Tree Sort: Calculando o tempo

- São n elementos inseridos.
- Cada elemento inserido percorre a altura da árvore.
- No pior caso a altura da árvore é n (sequência já ordenada) $\rightarrow O(n^2)$
- No melhor caso a árvore binária é completa, a altura é $\log n \rightarrow O(n \log n)$
- No caso médio a altura da árvore é proporcional a $\log n \rightarrow O(n \log n)$
- A varredura visita um item por vez, logo é $O(n)$.
- O algoritmo BST_Sort é $O(n \log n)$ no caso médio.

Heap Sort - A técnica

- O algoritmo de ordenação por Seleção em estrutura Heap segue as sequintes etapas:
  1. Primeiro construimos um Heap Máximo.
  2. O primeiro elemento é o maior da sequência. Trocamos este com o último, o último já está na posição. Vamos considerar agora a sequência sem este elemento.
  3. Aplicamos um Max-Heapfy nos $n - 1$ elementos restantes e o elemento pequeno que ficou no raiz é jogado para uma folha.
  4. Repete-se o processo até que sobre somente um elemento na sequência considerada.

Heap Sort: Max_Heapfy

Algoritmo $\textsc{Max\_Heapfy}$(A,n,i)

$e \leftarrow 2 * i$

$d \leftarrow 2 * i + 1$

se $e \leqslant n$ **e** $A[e] > A[i]$ **então**

    $max \leftarrow e$

senão

    $max \leftarrow i$

se $d \leqslant n$ **e** $A[d] > A[max]$ **então**

    $max \leftarrow d$

se $max \neq i$ **então**

    $A[i] \leftrightarrow A[max]$

    Max_Heapfy(A, n, max)

Heap Sort: Build_Max_Heap

Algoritmo BUILD_MAX_HEAP(A,n)
  para $i \leftarrow \lfloor n/2 \rfloor$ **até** 1 **faça**
    Max_Heapfy(A, n, i)

Heap-Sort: O algoritmo

Algoritmo $\textsc{Heap\_Sort}(A, n)$
    Build_Max_Heap(A, n)
    para $i \leftarrow n$ **até** 2 **faça**
        $A[1] \leftrightarrow A[i]$
        Max_Heapfy$(A, i-1, 1)$

Heap-Sort: Calculando o tempo

- O algoritmo Max-Heapfy executa em $O(\log n)$, que é a altura da árvore.
- O algoritmo Build-Max-Heap possui um cálculo mais complexo de complexidade, mas é $O(n)$.
- O Algoritmo Heap-Sort aplica n vezes o Max-Heapfy, portanto é $O(n \log n)$.