Capítulo 8

Cota inferior para algoritmos baseados na comparação de chaves

Os algoritmos de ordenação vistos até aqui são baseados na comparação de chaves, ou seja, a operação básica destes algoritmos é a comparação entre elementos do vetor que se quer ordenar. Neste contexto, vimos que *Insertion Sort* e *Quicksort* têm, no pior caso, complexidade de tempo quadrática, *i.e.* estão em $\Theta(n^2)$. Já *Merge sort* é capaz de, no pior caso, ordenar um vetor com n elementos em tempo $\Theta(n.\lg n)$. A dúvida que surge naturalmente agora é: seria possível construir um algoritmo baseado na comparação de chaves que consiga ordenar um vetor com n elementos, no pior caso, realizando menos do que $c.n.\lg n$ comparações (para alguma constante positiva c? Ou seja, seria possível melhorar a cota $\Omega(n.\lg n)$?

Para responder esta pergunta, assumiremos que os n elementos $x_1, x_2, \ldots, x_n$ a serem ordenados são distintos. Além disto, o processo de ordenação será modelado por *árvores de decisão*, que são árvores binárias completas. Assim, para construirmos a árvore de decisão para um algoritmo A (baseado na comparação de chaves) que recebe como entrada um vetor com n elementos distintos, anotaremos seus nós internos com $i : j$, para $1 \leq i, j \leq n$, e cada folha com a permutação $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. A execução do algoritmo corresponde a um caminho da raiz até uma folha. Cada nó interno corresponde a comparação $a_i \leq a_j$ (a rigor $a_i < a_j$ já que os elementos são distintos), e a subárvore à esquerda indica as comparações subsequentes. A subárvore à direita indica as comparações subsequentes quando $a_i > a_j$, e uma folha indica que o algoritmo ordenou o vetor de forma que $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$. Como qualquer algoritmo de ordenação correto tem que ser capaz de produzir todas as permutações de uma entrada, cada uma das $n!$ possíveis permutações do vetor de tamanho n dado com entrada tem que aparecer como uma folha. O caminho mais longo da raiz de uma árvore de decisão até uma folha representa o número de comparações no pior caso do algoritmo em consideração. Ou seja, o número de comparações no pior caso de um algoritmo corresponde a altura de sua árvore de decisão. A cota inferior da altura de todas as possíveis árvores de decisão será então a cota inferior, no pior caso, para qualquer algoritmo de ordenaçãqo

baseado na comparação de chaves.

Teorema 8.0.1. *Qualquer algoritmo baseado na comparação de chaves requer* $\Omega(n. \lg n)$ *comparações no pior caso.*

Demonstração. Precisamos determinar a altura de uma árvore de decisão onde cada permutação do vetor de entrada apareça como uma folha. Considere a árvore de decisão de altura h contendo l folhas de um algoritmo para ordenar n elementos distintos. Como cada uma das $n!$ permutações do vetor de entrada aparece como uma folha, temos que $n! \leq l$, e como uma árvore binária de altura h não possui mais do que 2^h folhas, temos que $n! \leq l \leq 2^h$. Portanto $h \geq \lg(n!) = \Omega(n. \lg n)$. □

Exercício 8.0.2. *Sejam l o número de folhas em uma árvore binária, e h sua altura. Prove que $l \leq 2^h$.*

Capítulo 9

Ordenação em Tempo Linear

Nesta seção veremos que se tivermos informações adicionais sobre os elementos a serem ordenados, podemos utilizar procedimentos que não são baseados na comparação de chaves para ordenar em tempo linear. Os algoritmos desta seção são conhecidos como algoritmos de classificação.

9.1 *Counting Sort*

O algoritmo *couting sort* assume que cada um dos n inteiros a serem ordenados estão no intervalo de 0 a k. Veremos que quando $k = O(n)$, a ordenação é feita em tempo $\Theta(n)$.

Algorithm 14: Counting-Sort(A, B, k)

```
1  let C[0..k] be a new array;
2  for i = 1 to k do
3  |  C[i] ← 0;
4  end
5  for j = 1 to A.length do
6  |  C[A[j]] ← C[A[j]] + 1;
7  end
8  for i = 1 to k do
9  |  C[i] ← C[i] + C[i − 1];
10 end
11 for j = A.length downto 1 do
12 |  B[C[A[j]]] ← A[j];
13 |  C[A[j]] ← C[A[j]] − 1;
14 end
```

Agora observe que o **for** das linhas 2-4 é executado em tempo $\Theta(k)$, o das linhas 5-7 em tempo $\Theta(n)$, o das linhas 8-10 em tempo $\Theta(k)$, e por fim o das linhas 11-14 em tempo $\Theta(k)$, o que perfaz um total de $\Theta(k + n)$. Assumindo que $k = O(n)$, temos que o tempo de execução de *counting sort* é $\Theta(n)$.

Definição 9.1.1. *Um algoritmo de ordenação é dito* estável *se não altera a posição relativa dos elementos que têm o mesmo valor.*

Exercício 9.1.2. *Prove que o algoritmo* counting sort *é estável.*

Exercício 9.1.3. *Prove que o algoritmo* merge sort *é estável.*

9.2 *Radix Sort*

O algoritmo *radix sort* ordena uma sequência de inteiros com d dígitos cada, em tempo linear. Ele utiliza um algoritmo auxiliar, que precisa ser estável, para ordenar a sequência de inteiros do dígito menos significativo para o mais significativo. Utilizaremos *counting sort* como algoritmo auxiliar.

Algorithm 15: radix-sort(A, d)

```
1 for i = 1 to d do
2 |   use a stable sort to sort array A on digit i;
3 end
```

Lema 9.2.1. *Dados n números com d dígitos, que por sua vez podem assumir até k valores, radix-sort ordena corretamente estes números em tempo $\Theta d.(n + k)$, se o algoritmo auxiliar estável tem complexidade de tempo $\Theta(n + k)$.*

Exercício 9.2.2. *Mostre como podemos ordenar n inteiros contidos no intervalo de 0 a $n^2 - 1$ em tempo linear, ou seja, em $O(n)$.*

Demonstração. Inicialmente iteramos pela lista dos números, convertendo cada um deles para a base n. Depois aplicamos o algoritmo *radix sort* sobre a lista, utilizando o *counting sort* como o algoritmo de ordenação dos dígitos da lista de números.

Dada a nova base, cada número possuirá 2 dígitos, uma vez que $\log_n(n^2) = 2$, onde cada dígito estará em um intervalo entre 0 e $n - 1$. Sabendo disso, vemos que *radix sort* ordernará n números de 2 dígitos, usando o *counting sort* em um intervalo de 0 à $n - 1$, resultando em uma complexidade de $O(2(n + n)) = O(n)$ □

Exercício 9.2.3. *Prove que o algoritmo* radix sort *é correto.*

Exercício 9.2.4. *Faça a análise temporal do algoritmo* radix sort*.*